Lessons from Three Years of Inspection Data

EDWARD F. WELLER, Bull HN Information Systems

In April 1990, Bull HN Information System's Major Systems Division began an inspection program as part of the standard development process for its GCOS 8 operating system. Data collection was crucial to the program's early acceptance, and the data collected over the last three years has been used to highlight potential problems and direct efforts toward continuous process improvement.

The project team's experience with the inspection program has given the development staff much insight into both the benefits and drawbacks of the inspection process. The lessons we learned were based on the use of metrics collected from more than 6,000 inspection meetings.

The GCOS 8 operating system is the descendent of the GECOS 3 operating system developed more than 25 years ago for General Electric Corp.'s 625/635 mainframe systems. It currently runs on the DPS 9000 series of mainframes.

“GCOS 8” collectively refers to all the system software, firmware, and test and diagnostic software—a total of more than 11 million lines of source code. About 400,000 to 600,000 lines of code are added annually, depending on the release cycle.

Table 1 summarizes inspection data for the last two-plus years. The increasing number of defects between 1991 and 1992 suggests that inspections became more effective because the number of defects removed increased faster than the number of meetings or size of material inspected. (Here, defect means major defects that would result in a visible failure if not corrected.) One reason could be that the work-product type has not changed, which means the inspection process has had a chance to stabilize.

In this article, I describe the methods used to collect and analyze the data and provide feedback to the development
staff. I also examine the data from the point of view of the entire organization and as it applies in individual case studies of particular project types. It is my hope that our experience and the lessons we have learned will help others better understand the usefulness of the inspection process as a tool for improving software-development quality and productivity.

**ORGANIZATION AND DATA COLLECTION**

Our company's software-engineering process group maintains records of data collected by project-inspection coordinators, who also perform a data-sanity and integrity check before passing it to the group. In a data-sanity check, the coordinators make sure attributes reported, such as number of lines of code, are indeed correct. In a data-integrity check, they make sure that the correct number of defects have been reported, for example, or that defects have not been labeled minor when they are major.

Data is collected in several ways. Projects may collect and analyze their data before forwarding it to the process group. In other parts of the organization, the data is given to project-inspection coordinators or to the secretarial staff for data entry before it is forwarded to the process group. This leads to a "hierarchical" collection of data, which can obscure the source of defects by the time the process group sees it.

When the process group does finally receive the data, they, along with inspection coordinators and other software-process engineers, analyze the data using a variety of PC-based tools and Bull's mainframe relational database. Interel includes information about size (in thousands of lines of code) from our configuration-management system, as well as test and field-defect data. Although this data is not yet fully integrated in the database, this combination lets us track product or module metrics beyond the inspection stage. This, in turn, lets us correlate data on defects at the shipping stage with inspection records for those defects.

**Training.** Training consists mostly of consultants providing inspection classes to all members of our technical staff, many first- and second-level managers, and members of our product-line-management group. These classes include a standard inspection-process class that covers the Fagan Method, a process developed by Michael Fagan to examine work products for defects. There is also a class tailored to the preparation and inspection of requirements documents. The project inspection coordinators hold weekly meetings, at which they discuss inspection-process improvements and inspection metrics. These meetings are also a form of training.

**Inspection metrics.** We collect the usual inspection metrics: time to prepare for the inspection, time to conduct the inspection, size of material inspected, and data on any defects. We also record a significant amount of data about the work product being inspected, which lets us conduct a causal analysis of the defects. We can then correlate the inspection data with projects, test data, and field data. Some of the metrics and identifiers are:

- **Inspection type.** What type of work product is it? Was the work product tested before inspection? Is this a first-time inspection or a reinspection?
- **Process.** Who moderated the inspection? What were the details of the inspection meeting (like time)? How large was the inspection team? What were the preparation and inspection times? What was the size of the material inspected? Were there changes in the material inspected, or was new material added?
- **Causal analysis.** What is the module ID? What are details about the defect? How much time was spent in rework? Causal analysis also includes failure reports — like Internal Software Notes, which report test errors, and System Technical Action Requests, which report errors after the work product is shipped.

**Collection method.** We began collecting data using paper forms. For the first six months, process engineers responsible for the inspection program reviewed all inspection reports before they entered the data into the database. Having engineers enter data also gave us a way to check it one last time before entry.

Early on, we recognized that the amount of data would become unwieldy as the inspection process expanded throughout the site. We developed a PC-based database program to retain the inspection records, including a tool that lets us collect data by having:

- a recorder — who could be a team member responsible for logging the defects — enter the data on a PC during the inspection meeting,
- the recorder or project-inspection coordinator enter data from paper forms after the meeting, and

**TABLE 1 SUMMARY OF INSPECTION DATA FROM 1990 TO 1992**

<table>
<thead>
<tr>
<th>Data category</th>
<th>1990</th>
<th>1991</th>
<th>1992</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code-inspection meetings</td>
<td>1,500</td>
<td>2,431</td>
<td>2,823</td>
</tr>
<tr>
<td>Document-inspection meetings (anything other than code inspection)</td>
<td>54</td>
<td>257</td>
<td>348</td>
</tr>
<tr>
<td>Design-document pages inspected</td>
<td>1,194</td>
<td>5,419</td>
<td>6,870</td>
</tr>
<tr>
<td>Defects removed</td>
<td>2,205</td>
<td>3,703</td>
<td>5,649</td>
</tr>
</tbody>
</table>

*The program started in May 1990, so figures represent May to December 1990.*
the size of inspected material (in our case, size of the module inspected) and defect severity. Errors in the first category are due to confusion over “size of material inspected” versus “new and changed material” (lines added or changed and then inspected) and clerical errors.

It is hard to discover these input errors once the data has been merged at the department level because the hierarchy of reporting tends to blur the errors of individual projects. On more than one occasion, we have found that higher-than-desired inspection and preparation rates are the result of bad records. To avoid these errors in the future, we plan to add data sanity checks to the collection tool for both size and rates. Meanwhile, we have added a reporting capability that provides preparation and inspection rates, defect rate, and inspection hours per defect to the collection tool so that recorders and coordinators can become more proficient at catching entry errors.

Lesson 1 about data collection is that you may have to sacrifice some data accuracy to make data collection easier.

The two entries most often in error are the size of inspected material (in our case, size of the module inspected) and defect severity. Errors in the first category are due to confusion over “size of material inspected” versus “new and changed material” (lines added or changed and then inspected) and clerical errors.

It is hard to discover these input errors once the data has been merged at the department level because the hierarchy of reporting tends to blur the errors of individual projects. On more than one occasion, we have found that higher-than-desired inspection and preparation rates are the result of bad records. To avoid these errors in the future, we plan to add data sanity checks to the collection tool for both size and rates. Meanwhile, we have added a reporting capability that provides preparation and inspection rates, defect rate, and inspection hours per defect to the collection tool so that recorders and coordinators can become more proficient at catching entry errors.

Lesson 1 about data collection is that you may have to sacrifice some data accuracy to make data collection easier.

When you collect data, avoid loaded terms or those similar to existing ones.

- secretaries enter data from paper forms.
- Most inspection teams prefer the last two methods, since the recorder is generally too distracted with entering the results on the PC to effectively participate in the inspection meeting. The last two methods are not without disadvantages, however. Errors are sometimes introduced, which necessitates a database search for inaccurate entries. The secretaries are not sufficiently skilled to check data integrity. But these disadvantages are not enough to overcome team members’ natural resistance to paperwork—a syndrome typical of most metrics programs.

Lesson 1 about data collection is that you may have to sacrifice some data accuracy to make data collection easier.

The two entries most often in error are the size of inspected material (in our case, size of the module inspected) and defect severity. Errors in the first category are due to confusion over “size of material inspected” versus “new and changed material” (lines added or changed and then inspected) and clerical errors.

It is hard to discover these input errors once the data has been merged at the department level because the hierarchy of reporting tends to blur the errors of individual projects. On more than one occasion, we have found that higher-than-desired inspection and preparation rates are the result of bad records. To avoid these errors in the future, we plan to add data sanity checks to the collection tool for both size and rates. Meanwhile, we have added a reporting capability that provides preparation and inspection rates, defect rate, and inspection hours per defect to the collection tool so that recorders and coordinators can become more proficient at catching entry errors.

Lesson 1 about data collection is that you may have to sacrifice some data accuracy to make data collection easier.

The two entries most often in error are

- A general confusion about how difficult the defect is to fix with how severe it is, and
- A misunderstanding about “severity” as the customer sees it or as it is defined in the context of testing.

To clarify the confusion and misunderstanding surrounding the terms “major” and “minor,” we recommend that projects just beginning an inspection process change the terms to “visible to the user” and “not visible to the user.” This is less threatening, eliminates confusion about how difficult the defect is to fix, and is (at least this was true in our case) not confusing with prior terminology. We do not recommend that projects switch terms in midstream, of course.

Lesson 2 about data collection is that you may have to sacrifice some data accuracy to make data collection easier.

Data classification. We introduced inspections in the middle of a major system release, while coding and testing activities were underway. Initial inspection metrics indicated widely divergent defect-detection rates. By classifying them according to the development stage in which inspection took place, we found a much greater consistency in defect rate. We used the following code classifications:

- Code inspected before unit test (new code).
- Reinspections.
- Code inspected after unit test.
- Fixes to problems reported during test or by the customers.

For our document inspections (which we did in addition to code inspections), we found 16 document types—two types of requirements, high- and low-level design, three types of test plans, three types of test cases, three types of test specifications (unit, integration, and system), project plans, user manuals, and process documents. The inspection metrics in these groups were sufficiently different to warrant the classifications.

Data analysis.

The divergent metrics we found before we classified the data could be attributed to a number of factors that have to do with organization-wide trends and attitudes toward inspection. These include reluctance to inspect code before testing, inspection-team size and effectiveness, the variety of high-level design languages, and process stability.
Inspection before test. As Figure 1 shows, the defect-detection rates before and after unit test differ considerably.

Code inspection after unit test continues to be more popular even though data has proved that this method of inspection catches far fewer defects. Glen Russell has identified two reasons for not testing before inspections. The first has to do with motivation. The inspection process relies on a motivated team, and testing creates the illusion that the product works. The second has to do with rework. When testing is extensive and the code producer has already logged many hours in creating the product, he may be unwilling to accept optimizations that would require major rework.

We found Russell’s first reason to be true in several of our other projects, which inspected after unit test and had a lower defect-detection rate during that inspection, while also having more defects after shipping.

Thus, there are four basic disadvantages of inspecting after unit test:

- Unit test lowers the motivation of the inspection team to find defects because it gives inspectors false confidence in the product. They “know the product works,” so why inspect?
- When large amounts of code are involved, optimizations can be very disruptive because all the code segments must be tested together in a unit test. When you inspect before unit test, you can generally inspect smaller work segments earlier in development. When inspections are done after unit test, the inspection team or project manager may be tempted to say, “We can’t inspect all this code now. We’re ready to enter integration test.” We faced this problem shortly after we began the inspection program, when we ended up doing an inspection after unit test. About 15,000 lines of batched code were waiting to be inspected, and it was very hard not to simply bypass the inspection process.
- When you do unit test first, you miss the opportunity to save time and resources. Sometimes, if inspection results are good enough, you can bypass unit test altogether and do a more economical integration test.
- Code inspections may uncover an underlying design defect not detected in design inspections, possibly resulting in considerable work if the inspections are postponed until after unit test. For example, in some of our new-code inspections, we discovered a serious design error months before the product was due to enter test, which let us recover without significantly altering the schedule.

The lesson is inspect before unit test.

Inspection teams. We investigated the effect of inspection-team size on defect-detection rates using data from more than 400 inspections that discovered defects. We found that four-person teams were twice as effective, and more than twice as efficient as three-person teams, which we attributed to two causes. First, the code producer is one third of the three-person team, but only one fourth of a four-person team. Second, a fourth inspector is a good indication that team members as a whole have a higher level of product knowledge.

Inspection teams require at least three people, so the only valid reason for adding a fourth inspector is to increase the team’s level of expertise. Three-person inspection teams may have members with a lower level of product knowledge because they may be serving as bodies to fill a quota.

Figure 2 shows two relationships of defect-detection rates versus preparation rates (the rate at which code was reviewed during individual preparation). The data shows three- and four-person code inspections, with each group further divided into preparation rates of greater than and less than 200 lines per hour. The four-person teams always outperform the three-person teams, and teams with lower preparation rates detect more defects.

Interestingly, a three-person inspection team with a low preparation rate seems to do as well as a four-person inspection team with a high rate. This may be because one person has a very high preparation rate, not necessarily because everyone has a slightly higher rate. The preparation-rate analysis supports our earlier supposition that four team members have a greater overall product knowledge than three members.

The lesson about team effectiveness is an inspection team’s effectiveness and efficiency depend on how familiar they are with the product and what their inspection-preparation rate is. Project managers must consider these factors as they develop their resource and schedule plans.

Design-document data. We have had more difficulty analyzing data from the inspection of documents than from code inspections. There are so many design methodologies and types of work—structured analysis and design, various program-design languages, and types of documents than from code inspections.
and structured or unstructured English—that we have insufficient data on each method to do a complete analysis, although we are recording the language used for these documents.

Figure 3 shows data from high-level design inspections. Defects per document page have changed over time, inspection size appears to be under better control (the "knee" in the scatter points has moved from 20 to 15 pages, and the defect-detection rate is higher), and inspection yield (number of defects detected) has improved. There are two possible explanations for the higher defect-detection rate. The first is that we were able to clarify the definition of defect severity during our start-up phase ("major defect" not meaning a defect that is complex or hard to fix, but rather one that is visible to the user). The second is that we were able to better control the amount of material inspected in one meeting.

The lesson (one that revisits lesson 2 under data collection) is ensure that the definitions of the metrics are clearly understood, and that guidelines for size of material inspected are followed.

Process stability. We have used the overall defect-density rates to measure the stability of the inspection process. Figure 4 plots the relative code and document-inspection yields for three years.

The dip in the first half of 1991 is due to a transition factor that occurred at that time. The initial data was provided primarily by a small number of pilot projects that received special attention from the process group and senior management. This group had lots of support for any changes and a suitable work product. As the process was applied to the entire site in late 1990 and 1991, the process group support was focused on the pilot projects was diluted across more projects.

To combat this dilution, I and later my colleague Robin Fulford provided the project-inspection coordinators with a data analysis of inspection and preparation rate, size of material, and other process metrics. Because we used actual data to emphasize the relationship of process metrics to detection rates, inspection teams were able to better understand why it was important to adhere to the recommended process metrics. As Figure 4 shows, the trend reversed and stabilized in the second half of 1991.

The lesson is expect a drop in effectiveness as you transition from pilot to general use, and be prepared to address the problems with process metrics.

CASE STUDIES

An organization-wide analysis of inspection data can help you assess the health of the inspection process, but the best use of inspection data is at the project level. Because the data is more likely to be from work on a common design, and from a more consistent set of inspectors, data analysis provides more useful information and guidance to the project members.

The case studies that follow demonstrate both the pluses and minuses of the inspection process. In some cases, the literal interpretation of the data will be incorrect. All four case studies presented are drawn from our code-inspection data.

Project A. This project illustrates how the inspection team can use metrics to evaluate the process. One of our pilot projects consisted of having inspectors collect and use additional metrics to measure their inspection effectiveness. By developing control limits that represented the norm for their inspections, the inspectors could measure each inspection against the expected preparation and inspection rates, yield in defects per thousand lines of code, and cost per major defect. They also kept reasonably accurate defect rates from unit tests.

The benefit of collecting this data, and understanding its value, was demonstrated during unit test. The code generated by the C compiler for one of the control microprocessors was inefficient and had to be replaced with Forth for several timing-critical routines. The team initially decided not to inspect the rewritten code (just do a simple conversion of 1,200 lines...
of C!), but testing the rewritten code was taking about six hours per failure. Because they knew they had been finding defects in inspections at a cost of 1.43 hours per defect, the team stopped testing and inspected the rewritten code. These inspections initially found defects at a cost of less than one hour per defect.

The defect density dropped significantly after the third inspection meeting, indicating the causal analysis of the defect source had eliminated the defects. Without the data that showed the relative costs, it would have been harder to back off the unit test. The team would have been fighting management pressure to continue debugging, as well as the natural resistance to inspection in favor of debugging.

Data at the end of unit test indicated that the code inspection before test was 80 percent effective (defects found by inspection/total number of defects). Effectiveness after system test was around 70 percent (inspections found 70 percent of all the defects detected after code was completed).

The lesson is get the data into the user's hands and they will use it.

Project B. This project, another one of the first to use code inspections, illustrates that good inspection results do not automatically mean the project will succeed. In this project, coding started just as the first inspection workshop was being offered. Consequently, the requirements and design documents had not been inspected. The project was also a prototype for a technology we had not previously used, which greatly complicated the problem. The team would have been able to find the design errors much more easily had they been familiar with the technology.

We inspected more than 12,000 lines of C, with an average defect density of 23 defects per thousand lines of code. The project was strongly motivated to perform inspections, and the project leader kept good records of each inspection. Indeed, in an analysis of 56 code inspections, both the preparation and inspection rates were within one standard deviation, indicating that the inspection process was under control.

With this kind of inspection data, you would think that the project would be successful. In fact, the team was given special mention during project reviews for their use of the inspection process. However, during the code inspections, the project leader observed that the code inspections were assuming a correct design, an assumption that may not have been warranted, since the team had not inspected the design documents.

The project leader's misgivings were well-founded. The project was not successful, and in a postmortem review, the team used the inspection data to pinpoint one of the key problems: Code growth during unit and integration test was almost 100 percent because of the continually unfolding requirements.

A comparison of defect data gathered during inspections and unit test showed that inspection defects were mostly coding or low-level design errors; defects found in test, on the other hand, were requirements and architectural-design defects, which had been primarily responsible for code growth.

Although code growth had not really been a surprise to the development team or management, the underlying reasons for that growth were not well understood. The detailed data highlighted specific design problems and let the development team base their conclusions on numbers and facts rather than conjecture and instinct.

Management decided to place the product on hold because the data showed that it clearly lacked the desired quality. The data was also used to justify process changes for future projects involving unfamiliar technology.

The lesson is good inspection results can create false confidence. Inspections are not a silver bullet. Be sure to inspect all basic design documents.

Project C. This project illustrates how inspection can improve the quality of
maintenance fixes. Repair of field defects is one of the most error-prone activities in software development; repair-defect rates can be as high as 50 percent for one-line changes, 75 percent for changes involving five lines, decreasing to 35 percent for changes involving 20 lines.3

Before we instituted inspections, we had collected the defective fix rate — the number of fixes replaced because of failures in the original fix — for almost one year in an attempt to improve the reliability of our fixes.

In August 1990, inspections for all fixes became mandatory. Figure 5, which depicts monthly defect rates in fixes to shipped products, shows the percentage of defective fixes dropped to about a third of what it was about this time. In the last year, the improvement has dropped to about half. (In September 1991, there were few fixes, so the small number of defects for that month created an outlier. Also, the Hawthorne effect — something being monitored improves simply because it is being observed — might have contributed to lower curves in the first half of 1992.)

When we correlated the reduced number of replaced fixes with the inspection data, we discovered an interesting anomaly. Only 13 percent of the inspections had detected defects, much less than the observed improvement for the first year, as measured by customer use. In other words, we could attribute only half the initial improvement to the defects found by inspections. We speculate that the inspection process could have had the initial side benefit of improving product quality independently of any defects the inspection uncovered. The initial improvement might also have been due to the Hawthorne effect. To determine its cause, we are currently examining the data in more detail to isolate possibilities, such as the size of the changes and any shift in the source of the defects. Without the causal-analysis metrics described earlier, this detective work would be impossible.

The lesson is inspections can improve the quality of maintenance fixes.

Project D. This project illustrates that inspections can be effective even when the data seems to indicate otherwise.

In this project, we were to inspect a new data-access service module and 112 modified modules with which it interfaced. In the 18,000 lines of code we inspected, we found 265 major defects. Table 2 shows the inspection and defect rates for this code.

Looking at the data in isolation, and observing the variance in inspection rates, it might seem natural to question the effectiveness of the new-service-module inspection (because the inspection rate was high and few defects were found). In fact, although we expected a higher defect rate in test, both inspections were equally effective.

A closer investigation of the new-service-module inspection data explained the high rate and low yield. The design used a large table of data that was well structured, which let us rapidly inspect a large segment of code and consequently have a lower defect density. The different rates indicated that the restructured design and code had fewer defects than the modified code.

Lesson 1 in this project is investigate the work product before deciding that process metrics indicate an ineffective inspection process.

Inspection data from this project provided another benefit. After the development team did a unit test on the first six modified modules without discovering any defects, they skipped the unit test of the remaining modules because they believed the inspection results warranted it. Consequently, they saved significant resources and time because unit test of this part of the operating system is lengthy and the test cases are run in integration test. Their decision was a trade-off of time saved in one activity against the potentially higher costs of defects in the next activity. The trade-off would be a good one only if the second activity had a small number of defects. The decision turned out to be wise; only four defects have been discovered in a year of unit test, integration test, and continued use of the product by other projects, against the 298 defects found by inspections.

Although this 98.7-percent inspection effectiveness (298/302) will undoubtedly drop during system test, the results show the power of inspections to influence the course of development.

Frank Ackerman and colleagues have suggested that inspections can replace unit testing, but not later stages of testing, and that benefits in later stages are mainly productivity gains.4 This project supported these suggestions. We were able to execute the unit-test suite in integration test to ensure that inspections did not miss categories of defects that are difficult to detect by inspection. We also effectively bypassed unit test with this product, and were able to do a "big bang" module integration — test many changes at once — rather than the usual continuous integration — accumulate a few changes at a time. Our test cost was essentially the fixed cost of the test, or the cost of running the test pro-

<table>
<thead>
<tr>
<th>Type of code</th>
<th>Inspection rate (lines per hour)</th>
<th>Defect-detection rate (per thousand lines of code)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified modules</td>
<td>Less than 138</td>
<td>About 22 defects</td>
</tr>
<tr>
<td>New-service module</td>
<td>300</td>
<td>About 10 defects</td>
</tr>
</tbody>
</table>

Table 2: Inspection and Defect Rates for Project D Code

In one case, inspection results were good enough to skip unit test and do a big-bang module integration.
grams one time.

Lesson 2 from this project (one that underlines some points I made earlier) is inspections can replace unit test with significant cost savings.

Analysis. We can learn several things about the inspection process from these case studies.

- Inspections are instrument the development process. Inspection data can be used by the development staff in real time to make decisions about which process is best to follow. In project A, the users had access to the metrics, and in project D, the development staff changed the test process.

You can also use inspection data to better understand the dynamics of the development process and provide hard data for decision making. In project B, the data was used to determine that the project was not successful. In project C, data was used to determine causes for a recent increase in the defect rate.

Inspection data collected from earlier development activities, from requirements through low-level design, has provided measurable indicators of progress during these activities. This hard evidence of progress has helped to forestall the inevitable query, “Why isn’t Sam coding yet?” because managers have a way to measure the completion and quality of the requirements and design work products.

- No matter how well they are executed, inspections cannot overcome serious flaws in the development process. In project B, the underlying faults were masked by the apparent progress reported by the inspection results. My advice to projects starting code inspections is to carefully evaluate the results. Providing data analysis to the development teams during inspection. We have had several successes when we fed the data back to the providers.

- Evaluating the level of detail or completeness of the design documents. In several cases, the high preparation rates and low detection rates are directly attributable to sparse design documentation.

Inspections can greatly improve software-development quality and productivity when users have first-hand access to the inspection data, understand the metrics and their effect on the inspection process, and use sound software-engineering methods and processes. When requirements are poorly understood, or subject to significant changes, the inspection process has little effect on the project’s eventual success or failure, although inspection data is valuable in postmortems.

The most pressing need in the future is to better understand data from requirements and design inspections. A key element required to compare defect densities and other rates is the ability to produce documents with consistent levels of detail across projects. We have just begun experimenting with $n$-fold inspections of requirements specifications — in which two or more teams inspect the same document — as a way to improve detection.

The inspection process is not perfect but it can prove a valuable tool for software developers and management in predicting a product’s quality and reliability.

ACKNOWLEDGEMENTS

Without the data provided by the software-development staff in Bull’s Major Systems Division and the efforts of Robin Fulford to collect and maintain the inspection database, this article would not have been possible.

I also thank the IEEE Software referees for their helpful comments.

REFERENCES


Edward E. Weller is a principal fellow at Bull HN Information Systems, Enterprise System Operations. His responsibilities include monitoring the software-development processes used by the Large System Software Products group, which developed GCOS 8, and activities to improve process ratings as measured by the SEI Capability Maturity Model. His prior interests are hardware design, diagnostic and design-verification software, and system maintainability design.

Weller received a BSEE from the University of Michigan and an MSEE from the Florida Institute of Technology. He is one of the cofounders and current cochair of the Software Inspection and Review Organization, a senior member of the IEEE, and a member of ACM and Sigma XL.

Address questions about this article to Weller at Bull HN Information Systems, 13430 N. Black Canyon Hwy., Phoenix, AZ 85029; Internet e.weller@bull.com.