

Introduction of Test Process Improvement and the Impact on the Organization

WAYNE D. WOODRUFF

Motorola Broadband Communication Sector

Testing organizations are often under the gun to test products, and they have little time to spend on process improvement. By developing and following a process improvement plan, an organization can make substantive improvement and improve productivity.

Often the mere mention of a process improvement project will instill fear into the team and kill the project. One effective way to avoid this common pitfall is to avoid making a commotion about the project and steer the team in small steps along the path to improvement.

Key words: inspections, process improvement, requirement management, traceability

INTRODUCTION

This article describes the process analysis and process improvement activities that took place over a four-year period in a software testing organization. The process improvement plan was based upon the Software Capability Maturity Model (SWCMM) (Paulk et al. 1993). This was found to be an excellent foundation on which to build our processes. As one will see, our improvement plan closely follows the Level 2 key process areas (KPAs) and at least one Level 3 KPA.

This test process improvement project was implemented by a test group that is part of a product development group (hardware and software). This product development group produces products used in the Cable-TV industry. My involvement in this project was as the manager of the test department that underwent these process improvement activities.

Initial Assessment

Using existing knowledge of the SWCMM, an informal assessment was performed. The assessment comprised comparisons between group activities and behaviors and the Level 2 KPA. No questionnaires were filled out and no formal results were presented. The assessment results were not shared with the group, nor were the improvement plans. While this flies in the face of accepted practices, reality is that when a person is confronted with change, there is usually a negative reaction to that

change. Rather than emphasize the plan and raise the fear level, it was decided to implement changes in small steps, almost covertly. Why announce a large process improvement project when the same thing can be accomplished by introducing subtle, but meaningful changes in a way that reduces or eliminates the fear and subsequent stress?

In a paper by Oliver Recklies (2001), a seven-stage model was described that depicts the changes a person goes through when confronted by change. By minimizing the fear factor, we skipped the first few steps in the model and put the staff through fewer emotional swings.

The main reason for this approach was that in a prior process improvement project the improvement plan was well publicized and became the center of attention. Many of the people involved in the project were reluctant to embrace the change concepts and felt the process was “being shoved down their throats” by a corporate mandate.

Rather than let history repeat itself, I decided to take a totally opposite approach in this project. This approach was to make subtle, incremental changes and celebrate the intermediate successes rather than make a big announcement, and present the details up front.

CONFIGURATION MANAGEMENT AND REQUIREMENT MANAGEMENT KPAs

Examining the test development side of the group, test specifications were written and informally reviewed by the test group. Little, if any, influence outside the test organization was brought in. Once complete, the specifications were placed under formal version control. From the test specification, tests were developed and integrated. Errors found in the test specifications during test development and integration were not routinely captured in a document revision. Essentially, once the document was under configuration management (CM) control, it was rarely revised. The impetus for updating the spec was usually the addition of a significant new feature. Major revisions rarely contained updates for previously identified problems, because the problems were not tracked. Test requirements were not clearly identified and tagged; consequently, there was no management of enumerated requirements.

For the test execution side of things, a proprietary tool was used for creating and executing test scripts. These test scripts were poorly managed as well. All machines that ran scripts were attached to LAN, and the scripts were run from a server. Because of network problems, the server was frequently unavailable. Since the testers needed to continue to make progress, they simply copied the scripts from the server to their local hard drives. Files were not systematically managed and the scripts on the machines often did not match the scripts on the server, nor was there any correlation between identically named scripts on the various machines.

Project Planning/Project Tracking and Oversight KPAs

Estimates were never generated for new test development; therefore, progress against plans was never tracked.

Regression test planning was instituted as soon as the project started. These plans identified the following:

- Code objects under test
- The testers responsible for execution of a particular subset of the regression suite
- The areas of functionality that were not being tested
- An estimate for the calendar time for test execution
- A list of risks
- Identification of where the test results could be found

The planning was a start, but it needed to be more accurate and contain more information for it to have any credibility.

Regression tracking was performed by keeping pass/fail status for all the tests in a spreadsheet. The tests were prioritized, and time estimates were kept for each test. Unfortunately, no process existed to capture and analyze actual test execution times, so the estimates were never revised and had no credibility. Likewise, there was no link between the plan and the actual test run. We had a nice plan that stated what we intended to do, but there was no way of verifying the plan was actually followed.

Software Quality Assurance KPA and Subcontract Management KPA

The organization did not have a separate SQA function, and we did not outsource any activities.

Assessment Summary

The results of the assessment were not pretty. Numerous problems were identified. Following are the most serious:

- It was impossible to understand how much effort a test cycle actually took.
- It was difficult to assess the status of the test cycle in real time.
- We had no empirical data from which to draw estimates for test development or test execution.
- When a test failed, the results could not necessarily be duplicated because the test script or environment had likely changed.
- The test status sheet was hard to read and understand.
- Certain tests had to be run by certain people. Little cross training was done. If a key person was out, that test had to wait until he or she returned.
- The only person who understood a particular suite of tests was the person who developed those tests.
- There was little control over the test configuration. There was no standard test station. The test environment was not controlled.
- Test requirements were not enumerated and managed.
- Errors found after the test specification was complete were not systematically corrected.
- The test organization had little credibility, and morale was poor.

From the management perspective, there were several key objectives of the process improvement project:

- To obtain up-to-date status of any test object under test at any time
- To accurately understand the effort to run components of the test cycles, so test effort could be predicted
- To be able to reliably reproduce test failures

- To improve group morale, and give the test team a sense of accomplishment

THE PLAN

Once the assessment was completed, a four-phase plan was devised. The idea was to fix the major problems and come back later for fine details. The four phases in the improvement project are:

1. Configuration management
2. Formal inspections
3. Requirements management
4. Planning to track/tracking to plan

One might ask, "Why these phases and this order?" It is my opinion that CM is the basis for all development. Without it, chaos will rule. Likewise, it is my opinion that inspections have a well-documented return on investment and provide numerous additional benefits. Requirements management requires a bit more discipline and maturity in the organization, so that was saved for later. Planning to track/tracking to plan was last because I believe that this particular phase requires the most maturity in an organization.

As the manager, I thought there were several critical success factors that were required for the team to succeed:

1. At the start of any phase, explain to the team the reason for a change. Explain the problem that the team is attempting to solve and the desired results. Be sure the desired results are obtainable.
2. Give the team training on any new tools for the current phase and allow the team adequate time to become familiar with the tools and processes as the project evolved. Training does not need to be formal; mentoring or on-the-job training is actually more desirable.
3. Do not mandate a solution. Provide guidelines. Allow the team to develop the solution.
4. Do not abandon the process when a crisis arises.
5. Make small but substantive changes.

By committing to the process, I tried to show the team that I believed in the process and the subsequent outcome. This seemed to inspire them to exceed the goal, rather than to think of it as a passing fad.

PHASE 1: CONFIGURATION MANAGEMENT

In the CM phase, all configuration items were identified. Configuration items included the test specifications, test scripts, test application source code, and the test environment. The team had felt the pain of the lack of CM, so they were ready to embrace change.

As discussed before, the test scripts were a mess and were addressed first. These were fairly simple to fix, but time consuming. The network infrastructure problems were quickly solved. The next task was to get the scripts under control of a commercially available CM tool. To perform that task, all the scripts had to be reviewed to determine the appropriate scripts to use as the configuration managed version. This was done as the scripts were used during test cycles, and it consumed the better part of four months as well as several regression cycles. Once this was completed, the testers were mentored on CM concepts and the tool. Scripts were checked into the CM tool, and the most current revisions of the scripts were sent to a well-known location on the network. The test environment was modified so that the test stations all used the same location for test scripts. The CM tool was used to handle ongoing maintenance of the scripts.

Another step in this process was to define a standard test station. Each section of the test suite was examined to determine if it had any special equipment requirements. All stations were identical, except for one station that had additional equipment for running one set of tests. Over time, the special dependency was removed and all test stations now have identical configurations.

Finally, the test environment was located in a laboratory that was also shared by development and test personnel. Often, the development personnel made changes to the lab that would affect testing. A cross-functional team of development and test people was formed and asked to define a shared environment so that development personnel had the ability to make lab changes that would no longer impact testing. Any subsequent changes made to the lab were now scrutinized for cross-functional impact.

These simple changes made a world of difference. Other aspects of CM, namely the management of test specifications and test requirements, were deferred to the requirement management phase, since they are so closely related.

PHASE 2: FORMAL INSPECTIONS

Just five months after this project started, the leaders of the test team were trained as inspection moderators. They were trained to use a Fagan-like process (Fagan 1986) and have used this process for more than four years. My unstated goal was to inspect 90 percent of our test specifications. It took discipline on the team's part, but the process became a part of our culture within a year, and to date, we have achieved 100 percent inspection of the test requirement specifications. The team members were reluctant at first to participate in an inspection, especially if they were the producer. I helped by being a participant in every inspection for the first two years and reassuring them that this was not a witch hunt. I also had the team inspect some documentation that I produced to show them how they could help me.

The inspection team usually consisted of five people and included the author of the document, also known as the producer, a software developer (preferable the one who designed the code), and at least one of the people responsible for running the test (testers). The other two team members were the moderator and recorder. The moderator runs the defect-logging meeting and the recorder records the results of both the defect-logging meeting and the individual inspections. Inspectors had at least a week to review the document and generate a list of defects using standard forms and checklists. The results were captured in a master defect list during a defect-logging meeting. The producer was given the master list and fixed the issues. Depending upon the nature of the defects, a reinspection may have been warranted; the inspection team was empowered to decide whether to reinspect the test requirements document.

Measurements were taken as part of the process and from those measurements the following metrics were computed:

- Inspection efficiency - minutes of inspection per defect found. This includes both inspection and logging time. For comparison sake, this is also computed in dollars.
- Defects per inspected page
- Cost to find a defect (historical)

Based on three major product developments and 98 inspections over the last four years, it takes only 17 minutes to find a defect in an inspection, and assuming a labor rate of \$100/hour, it costs us \$28 to find a defect.

Another notable benefit of the inspection process is that it offered the opportunity for the testers and developers to interact during the inspection, so that developers helped identify tests that may have been missed. The testers, the people who actually run the tests, also received advance notice of upcoming new tests. It allowed the test team the opportunity to identify diagnostic requirements so the software requirements could be validated. It was a great opportunity to train test staff, to share ideas, and to reinforce excellent working relationships between developers and testers.

PHASE 3: REQUIREMENTS MANAGEMENT

This phase introduced formal requirements management, but had strong ties to the inspection process and the CM process. This was a big change, since it involved the introduction of a new tool. This phase took the longest to institutionalize.

About a year after the process improvement project began, the company began to develop a major new product. Having had many years of experience with a commercially available requirements management tool, I convinced the software team to let the test team capture the software requirements in the tool. The software developers and testers needed some training on the tool for the management of requirements, but since the tool used the same word processor, it was fairly simple. Once the document was brought in to the tool, we allowed the tool to scan the document for a keyword and create a database of software requirements. For software requirements, we chose “shall” as the keyword and “shall verify” for test requirements.

For each software requirements specification (SRS), a test lead was assigned. That lead person analyzed the SRS and developed a test requirement specification. After a draft document was completed, the test lead would meet with the software engineer and try to clarify any software requirements we did not think were testable. Note that the software team does not use inspections, so their document was official after one or more reviews of the document. It was entirely possible that the SRS could be released with untestable requirements!

Once that effort was completed, a formal inspection of the test specification was conducted. After the test requirements specification was revised, it was captured in the requirements management tool and the database was augmented with the test requirements.

The next step was to trace the test requirements back to the software requirements. Again, the requirements management tool fully supported traceability, so it was a matter of allowing the team time to perform the traceability and subsequent analysis. This is one area that we must focus on for future improvement. The traceability should be included as part of the inspection process, not conducted after the inspection is complete.

Once traceability was completed, some key metrics were generated and reviewed. These include number of traced software requirements, number of untraced requirements, and ratio of test requirements to software requirements. Software requirements that did not have test requirements that were traced to them were investigated. Root-cause analysis of those untraced requirements showed that approximately 90 percent of them were attributed to being design requirements, which were not testable. As mentioned previously, the software requirements are reviewed but not inspected. The test group often found defects in the software requirements during the inspection of the test specification, as they were always used as a reference document for the test specification inspection.

One advantage of the requirements management tool was that it had a built-in CM that was better than the company’s existing CM tool. It kept revision information on every requirement as well as on each document and the project as a whole. Our CM tool would only maintain revision history on the project, so we chose to use the abilities of the requirements management tool.

Some test specifications were not based on software or hardware requirements. One example was our product that employed a proprietary communications interface. The protocol definition was external to the requirements repository. We had developed numerous test specifications for the protocol and updated our specifications when the protocol definition was expanded. While it would be nice to trace back to the protocol requirements, we manage without. The protocol specification is not frequently updated, so updates are very well documented, so we can easily establish the impact to the tests.

Likewise, there are numerous implicit requirements, based on a historical perspective, both product and industry. We handled them in the same manner. We developed test specifications and captured test requirements; they were not just traced back to originating hardware or software requirements. The policy was that every test we ran must be documented in a specification.

PHASE 4: PLANNING TO TRACK, TRACKING TO PLAN

The approach of this phase was to put together a simple and effective way of capturing test execution status. Later in the phase, the planning aspect fell out of that status reporting mechanism.

Our existing tracking spreadsheet was poor (see Figure 1). It had several worksheets: one for each major test area, and one for the summarized pass/fail status. The summary sheet was particularly confusing. Ran was defined as pass plus fail, so %ran was ran divided by number of tests. Total coverage was the sum of ran, can't test, and not tested. The "definition" of total coverage was vague and was ignored.

Looking at the detailed status sheet (see Figure 2), each major test area worksheet had a different format, which contributed to the confusion, but most worksheets had a core set of columns: test number, functional requirement, test case, test script, and pass/fail.

The *test number* was just a sequentially numbered value that was not traceable to any software requirement. The *functional requirement* was a description of the test. Unfortunately, its origin was thought to be a generic description entered by the test developer and was very subjective. The *test case* was a cryptic string of letters and numbers whose origin was unknown. The *test script* defined the script that the tester was to use. The *pass/fail* was the result the testers arrived at by running the test and had one of four possible test outcomes: pass, fail, can't test, and not tested.

Basically, the spreadsheet tracked the status of running the scripts. If a test had a status of other than "pass," there were not any notations of why it failed, could not be tested, or was not tested. If the test failed, there was no link to a defect report. Each time a test cycle was run, the same bug occurred. After a while, the tester who ran it knew which tests would not pass and stopped running tests. If the bug was fixed, the fix was never verified!

The requirements for improvement were that the status had to be easy for the testers to use to track detailed progress against, but provide planning and summary status for management. In the past, a document was generated with the plan for the regression cycle. There was no mechanism for verifying that the plan was actually executed, so the ideal tracking method would also include the planning information.

The fix was rather drastic. The first step was to create a consistent look and feel to the detailed status sheet. One of the test managers mulled this over for a few weeks and suggested that test requirements be exported from the repository and brought into individual status worksheets. By doing so, we captured the test requirement number and test requirement text from the database, along with the name of the application or script that was used to test this requirement. These are represented in the spreadsheet as TR# and test requirement, respectively. This solved a multitude of problems and the worksheets had a consistent look and feel.

A second effort was to create clear and consistent definitions for the status-reporting column. This

FIGURE 1 Old summary status

| | Test A | Test B | Test C | Test D | Test E | Not Supported | Totals |
|------------------------------|--------|--------|--------|--------|--------|---------------|--------|
| Number of tests | 52 | 179 | 119 | 15 | 26 | 10 | 401 |
| Ran | 51 | 168 | 94 | 12 | 0 | 0 | 325 |
| Pass | 48 | 163 | 94 | 12 | 0 | 0 | 317 |
| Fail | 3 | 5 | 0 | 0 | 0 | 0 | 8 |
| Can't Test | 1 | 1 | 2 | 3 | 0 | 0 | 40 |
| Not Tested | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Total | 52 | 179 | 119 | 15 | 0 | 0 | 365 |
| % RAN | 98.1% | 93.9% | 79.0% | 80.0% | 0.0% | 0.0% | 81.0% |
| Total Coverage % (Run+CT+NT) | 100.0% | 100.0% | 100.0% | 100.0% | 0.0% | 0.0% | 91.0% |

© 2003, ASQ

Introduction of Test Process Improvement and the Impact on the Organization

FIGURE 2 Old detailed status sheet

| Test # | Functional requirement | Test case | Test script | Pass/Fail |
|--------|--|--------------|-------------|-----------|
| 23 | The product will illuminate the front panel... | DCTACC-FW-04 | Xyx.txt | P |

© 2003, ASQ

required a few iterations as we realized that the existing test status definitions did not clearly state the result. Over time, the four grew to seven with the following definitions:

P—Pass

- Testing completed and expected results achieved.

F—Fail

- Testing completed and expected results were not achieved. A new defect must be written. To facilitate testing of future code releases, the tester enters the bug number from the defect tracking system and the code version into the Code CR (change request) column.

CT—Can't test (external problem)

- Lack of verification mechanism
- External equipment missing/broken
- Test developed, software functionality not supported

NT—Not tested (test group internal problem)

- Test script/app under development
- Test script/app needs maintenance

KI—Known issue

- An existing defect within another area of code, does not allow further testing to be completed.
- Test verification mechanism (that is, diagnostic output) requested but not implemented.

KF- Known failures

- Test fails because of an existing defect, which was not fixed in this release.

PV- Pass verified

- When retesting a known defect, and expected test results are achieved.

The following columns were added to the spreadsheet:

- Code CR – Identifies a defect report and is used when a test fails and there is a defect in the product.
- App/Script CR - Identifies a defect report and is used when a test fails and there is a defect in the test.
- Comments

The result is shown in Figure 3. Likewise, we added information to the summary table.

First, we added in the name of the tester who ran the major section and how many hours were spent on that test. Over the course of many test cycles, all testers had the opportunity to run each test suite several times. This served several purposes. All the testers became familiar with all of the test suites and the average execution time could be computed from the empirical data. Since the average execution time for each suite was well known, a very accurate schedule could be produced. This also allowed for accurate schedule generation in cases where resource levels fluctuated.

We added a column for “Testing Planned.” Normally, we run the complete regression cycles on each software release, so each entry would be “yes.” In the case where we decide to run a reduced regression cycle, we assess the changes to the software and what test will

FIGURE 3 New detailed status worksheet

| TR# | Test requirement | App/Script | P/F/CT/NT/KI/KF/PV | Code CR | App/Script CR | Comment |
|--------|--------------------------|------------|--------------------|-------------------|---------------|-------------------|
| TR1063 | Verify the xyz shall ... | Abc.txt | F | 1234-5678 1.21 | | |
| TR1064 | Verify the sun rises... | Qec.txt | KF | 1234-5660 1.19 | | Bug not yet fixed |
| ... | | | | | | |

© 2003, ASQ

Introduction of Test Process Improvement and the Impact on the Organization

FIGURE 4 New summary status information from spreadsheet

| Test area | Testing planned | # Test requirements | Ran | Pass | Fail | Can't test | Not tested | Known issues | Known failures | Tester | Time |
|---------------------|-----------------|---------------------|-----|------|------|------------|------------|--------------|----------------|--------|------|
| Area 1 | Yes | 35 | 35 | 32 | 2 | 1 | 0 | 0 | 0 | Jim | 3.5 |
| ... | | | | | | | | | | | |
| Area n | No | 45 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | |
| Total tests planned | | 437 | 429 | 412 | 15 | 1 | 0 | 1 | 0 | | 80.5 |
| | | | | | | | | | | | |
| New CRs found | 12 | | | | | | | | | | |
| Hours/new CR | 6.7 | | | | | | | | | | |

© 2003, ASQ

exercise those changes. Any tests that do not need to be run get marked with a “no.” Based on the “yes” entries in the testing planned column, the total number of test requirements is computed. Once the planning was completed, the testers printed the detailed worksheet (see Figure 3) for the areas they were going to test. The detailed spreadsheet contained all the information the tester needed to do his/her job. Each tester kept a paper copy and noted updates on this copy as he or she ran tests. Once a day, each tester used the data on the paper copy to update the electronic spreadsheet.

A worksheet was added to list any new defects found in a release. This allowed testers to easily share any new defects without the need to query the bug database. All testers knew to check the defect worksheet prior to submitting a bug so duplicates were not entered. The total number of bugs found were computed and added on the summary page.

Since we had new defects found and hours testing used, we computed how many hours of testing it takes to find a new defect. Empirical data tell us it takes between 14 and 30 hours of test time to find a defect during a regression cycle. Based on the \$100/hour rate, this translates to \$1400 to \$3000 to find a defect during regression testing. The dollar figure is not that notable, but the hours are. If we plan to spend 150 hours testing a new release, on average, the software team can expect at least five bugs from that effort. We call this metric “the cost to find a defect” metric. The resulting summary status page is shown in Figure 4.

We used this process for a short period of time, but it still wasn’t quite right. To create a status sheet for a new test cycle, we copied the results from the last cycle and deleted the pass/fail status. This allowed us to carry over the defects and comments from the prior cycle. The problem with this was if we added or modified any test requirements, our status sheet would not

contain those new or modified test requirements. The only way to accomplish this was to update the spreadsheet by entering/modifying test requirements in both the database and the spreadsheet. We needed to resolve these issues to make the process better.

We actually introduced two new improvements as a next step. Whenever a new defect was found, the tester enters the code defect number as well as a code version in which the defect was found in the requirements repository against the test requirement. This kept the requirement database current with new failures. Likewise, if a test that previously failed now passed, the tester removed the defect number from the requirements repository. In order to keep this from being a burden to the testers, they waited until the current test cycle was completed and made all the updates at one time.

Secondly, at the start of any test cycle, we exported the test requirements from the database and used them to create a new status tracking spreadsheet. By doing so, we corrected all the problems. The status sheet would contain the latest test requirements and the results from the prior test cycle.

This really closed the loop and guaranteed that each test cycle had the most current test requirements and defect history. A series of spreadsheet macros was developed that automated the building of the status sheet and made the process more user friendly. Now, we had the most current test requirements and the issues found in the last cycle were automatically carried over to the next cycle.

A separate spreadsheet was maintained for each software version tested. This was beneficial in that there was a snapshot of the exact tests run and the status of those tests for each version of code. As the test suite was enhanced over time, the history of the test evolution was saved.

THE RESULTS

After four years of hard work, benefits of the process are being enjoyed.

| Year of improvement project | Number of test cycles run | Percent increase from year two | Number of test requirements covered by a test cycle | Percent increase from year two |
|-----------------------------|---------------------------|--------------------------------|---|--------------------------------|
| 2 | 48 | N/A | 1600 | N/A |
| 3 | 48 | 0 | 1800 | 11 |
| 4 | 79 | 65 | 2640 | 65 |

More impressive is that we increased our productivity dramatically, without adding a single resource! As a matter of fact, at the end of year three, we *lost* a resource!

Our test cycle schedule accuracy is excellent. We know how long it will take to run the cycle and how many resources it will take. We rarely miss the deadline! We can pull the schedule in by adding resources (to a point). Overtime is the exception rather than the rule.

Next steps

There are two areas for us to focus on in the short term. First, we should start inspecting our test scripts and the source code for test applications. In the original concept of the plan, source code and script inspection was thought to be too expensive and we should focus on the upstream documentation first. After several years of inspection documents, the belief that code/script inspections are too expensive has passed.

Second, I would like to expand the planning to track/tracking to plan to include test development. We routinely develop schedules for test development, but we do not track it like we do a regression cycle. Granted, this is a considerably different process, but estimating and measuring our development will only help to improve those as well.

CONCLUSIONS

The project is a resounding success. I write, “is” because the project is not over and it never will be. Improvement is a journey, not a destination. Continuous improvement provides valuable insight to the organization and has fundamentally changed how we do our job. We have and will continue to make substantive improvements to how we do our job. Our team is very agile and has very predictable schedules.

- We now have great understanding of how much effort a test cycle actually requires.
- We have real-time test status.

- We have empirical data from which to draw estimates for test execution.
- We can readily duplicate test results.
- The test status sheet is much easier to read and understand.
- Cross training is a way of life. We are rarely bound to a single person owning a test.
- There is a standard test station. The test environment is tightly controlled.
- Test requirements are enumerated and managed.
- Errors found after the test specification was complete are systematically corrected.
- The test organization has a great sense of pride and accomplishment.

I'd like to reveal an interesting endnote. After developing this process for three years, I was commenting to one of my managers about how our process had improved. The manager gave me a puzzled look and asked me what I meant. I showed the manager a flow-chart of the process and described the process. After looking at it for about 30 seconds, the manager's eyes opened wider and the manager smiled and said, “I had no idea that we were following such a process.” Likewise, a few months ago, I was asked to present our process improvement project to corporate management. Afterward, I decided to present it to my staff. I got a similar comment from one of my senior developers after the presentation. While there are several ways to interpret this, my understanding of the response was that I had created an environment of continuous improvement, without people feeling burdened by the process.

REFERENCES

- Fagan, M. E. 1986. Advances in software inspections. *IEEE Transactions on Software Engineering* 12, no. 7 (July).
- Paulk, M., B. Curtis, M. B. Chrissis, and C. Weber. 1993. Capability Maturity Model for Software, version 1.1 (CMU/SEI-93-TR-024). Pittsburgh: Software Engineering Institute, Carnegie Mellon University.
- Recklies, O. 2001. *Managing change: Definition and phases in change processes*. See URL www.themanager.org/Strategy/Change_Phases.htm.

BIOGRAPHY

Wayne Woodruff is a senior manager for Motorola Broadband Communication Sector, where he is responsible for software development for Advanced Interactive Digital Cable Settop boxes. This article was written while he managed testing of Advanced Interactive Digital Cable Settop boxes. He has been working in embedded systems design, development, and testing for more than 23 years. Woodruff has a master of electrical engineering degree from Villanova University and a bachelor of electrical engineering technology degree from Spring Garden College. He has published numerous articles ranging from Embedded Systems Design to Integrated Development environments to Requirements Management, which can be viewed at <http://www.jtan.com/~wayne/publications.html>.

Woodruff is a member of ASQ, and he can be reached at wwoodruff@motorola.com