2009

PACIFIC NW

SOFTWARE QUALITY CONFERENCE

MOVING QUALITY FORWARD

October 27-28, 2009

Conference Paper Excerpt

From the

CONFERENCE PROCEEDINGS

Permission to copy, without fee, all or part of this material, except copyrighted material as noted, is granted provided that the copies are not made or distributed for commecial use.

Improving Your Quality Process: A Practical Example

John Balza

Software Quality Consultant and Instructor

johnjbalza@comcast.net

Biographical Sketch

John Balza currently is a software quality consultant and teacher working with software companies to improve their software quality using inspections, defect analysis and metrics. John has presented at numerous software quality conferences since 2003.

John was the Quality and Productivity Manager at Hewlett-Packard in Fort Collins, Colorado. John was responsible for overall product quality for an organization of 1500 engineers in six geographic areas developing HP's version of UNIX, HP-UX. In this role, John reduced customer defects by over 80% and improved productivity by 30%. These changes were made by gaining management sponsorship for key process improvements and then facilitating teams to make these improvements. This included creating management metrics to track quality throughout the lifecycle, using inspections and defect analysis throughout the life cycle, applying agile methods to large complex projects, and assisting the lab organizations to achieve level 2 and 3 of the Capability Maturity Model. Prior to this assignment, John had managed over 50 software projects at several levels of management.

Abstract

Organizations are often faced with the problem of improving their quality process and justifying the investments for inspections and improved testing. This paper will discuss how Hewlett-Packard developed an improved quality process by:

a) Understanding the types of defects that were escaping from the product,

b) Using a model for the costs of finding and fixing defects to select various quality assurance techniques, and then

c) Developing a quality process that included multiple inspections and test types.

Background

At the 2003 Pacific Northwest Software Quality Conference, I presented "Management Commitment to Quality Requires Measures"[i] that focused on the necessity to give management quality measures that were as available as schedule metrics. This paper explains the process side of that experience. It shares the business case we built for Hewlett-Packard's (HP) management to improve our quality and the changes we made in our quality process.

The HP-UX product is Hewlett-Packard's version of the Unix operating system. It consists of approximately 18 million lines of code produced in a distributed organization of 1500 people in 7 worldwide locations. These 1500 people are organized into 13 different R&D labs reporting into several division managers. The product has a major release every 3 years or so, with quarterly updates in between. The size of these major upgrades had been doubling on each release. Each lab is responsible for the quality of their code and then one lab is responsible for the system testing. The HP-UX 11.00 version of our product was released with significant functionality additions to the previous release; in particular it was our first 64-bit operating system. Many customers began to use the operating system in mission critical applications where availability and reliability were very important. In these new environments, customers were demanding higher levels of quality than in our previous releases. In a survey conducted by the HP User Group, "developing higher quality software" was the 2nd highest issue in the poll.

Given the customer situation, management decided we needed to set an aggressive goal to improve our quality by 10X over 5 years – that is we would reduce the number of defects found by customers by a factor of 10 in those 5 years. This goal was chosen for two reasons – a) it was a familiar goal to the organization – in the 80's a similar goal had been declared as a corporate goal, and b) 10X would get us to the same level of quality as a mainframe – so it was a good competitive goal.

In addition to the overall 10X goal, we decided we should have other goals that would help us achieve the long-term customer goal:

- Goal 1: subdivide the long-term goal. We chose an equivalent to 10X in 5 years, reducing customer defects by 2X every 18 months.
- Goal 2: reduce defects before they affect other engineers. We focused on reducing the defects before the code was checked-in. Check-in was the point where an engineer's code was delivered to everyone in the project.
- Goal 3: Improve our system testing to further reduce the defects delivered to customers. It was general knowledge at the time that our system testing no

longer reflected our customers' environments – the typical customer used to be an engineer's desktop, but now it was large database applications in telecomm, finance and manufacturing organizations.

Target Improvements Using Defect Analysis

If we were really going to improve our testing, we needed to understand what improvements were needed. We asked every subsystem to conduct a defect analysis on the defects that were escaping to system test and to customers. Then we required a root cause analysis of the most frequent defect types with recommendations on what change was needed in our quality process to remove the root cause. Defect analysis has a key advantage over just imposing better quality processes. The engineering teams analyze their own data to understand what types of defects are escaping and then they recommend how to reduce these escapes.

Defect analysis[ii] is pretty simple. First find the most common types of defects. I recommended doing Pareto charts based on the lifecycle phase the defect was introduced, the type of defect (algorithm, locking errors, non-initialized variables, etc.) and the module where the error occurs. Take the most common types and figure out the root cause of the defect, (ask WHY 5 times) and then make a recommendation on how to remove that root cause. We found one consistent problem: often the first recommendations are very generic, like "better reviews" or "better testing". We required specifics: what should be done in the review, what types of tests need to be added, what training was needed.

After having each subsystem complete their recommendations, there were some common themes across the product that stood out.

First, the majority of defects were in the core operating system (what we call NCKL, Networking, Commands, Kernel, and Libraries). Historically, we had built our product inside out – putting together the core, testing it, and then layering other areas (user interfaces, system management software, and other middleware) on top of it. We would spend months getting the core to be stable enough to layer the other software.

50% of the defects were functional defects – things that should have been found by good functional or black box testing.

Another 10% were defects that only showed up under certain configurations. Engineers didn't have access to many of the configurations where problems occurred – at best these would be found in system testing, at worst, customers would find them.

Finally, we noted that developers only found about 30-40% of their own defects – most were being found by other development engineers or system test and 12% were found by customers. This was a very inefficient way to find defects.

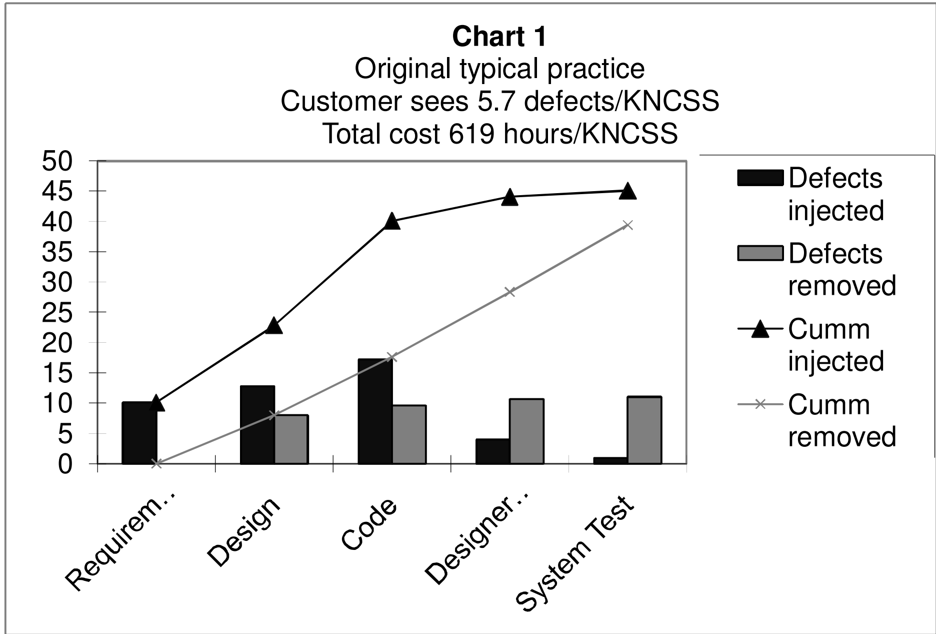
The Business Case for Management

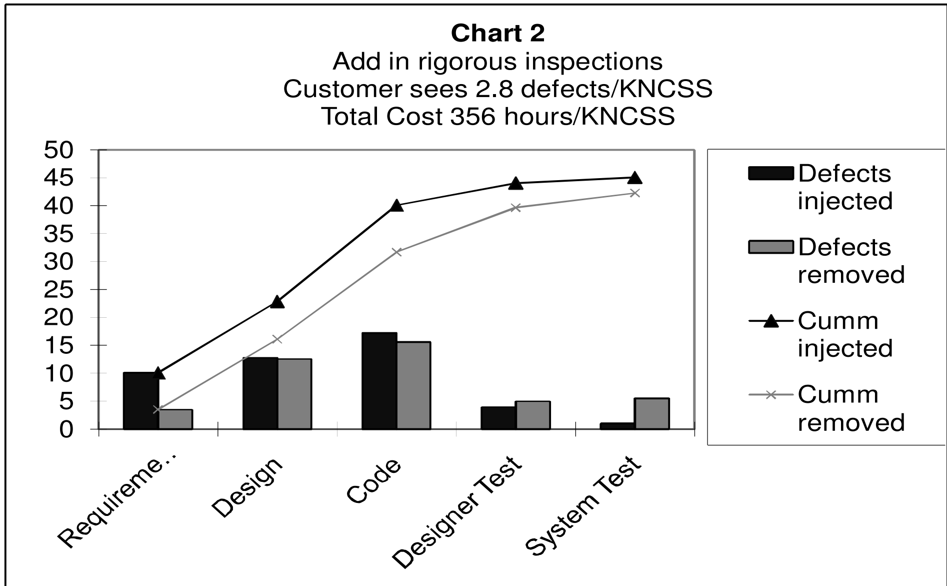
Of course, if we were going to make major changes, we needed to show our management team that these changes would actually save them money. What follows are the actual justification slides that I used at the time. First, I used data from Capers Jones[iii] to demonstrate that half of all defects are introduced before coding begins. Second, Capers also publishes the typical efficiency for various peer reviews, inspections, and test methods. One key point of his data shows that formal inspections are twice as efficient as informal review. For example, the median removal rate for a design review is that it will find 30% of the defects in the design, while an inspection will find 65% of the defects. New function testing will typically remove 30% of the existing defects. Of course, that new test will eventually become a regression test. Regression tests only find 15% of the remaining defects. This is to be expected – a new test tends to find new defects, rerunning that same test will only find whether a change elsewhere in the system broke something. Finally, we used a combination of our in-house data and Capers Jones[iv] data to determine the typical hours it took to find and then fix a defect with each of these processes.

Using the data of the number of defects introduced and the time to find and fix a defect, and knowing our typical quality methods of the day: a design walk-through, a code check, writing new function tests, running existing regression tests, and having a system test, resulted in Chart 1. It shows the number of defects injected during each phase, and the number removed. The lines show the cumulative defects injected and removed. And, of course, the difference between the two lines shows the number of defects remaining in the code.

So given our current methodology, we were spending about 619 hours finding and fixing defects and shipping with about 6 defects per thousand lines of new code. Since the time to find and fix a customer found defect was well over 40 hours, not counting field time, shipping that many defects to customers was extremely costly.

Chart 2 looks at inspecting everything (remember we typically had only design walkthroughs and code desk checks). What was the most convincing was that we could reduce the defects shipped customers by a factor of 2, while saving 42% of the labor. Most of the labor savings come from the fact it typically only takes ½ hour to fix a defect in an inspection versus multiple weeks if a defect is found by system test or by the customer.

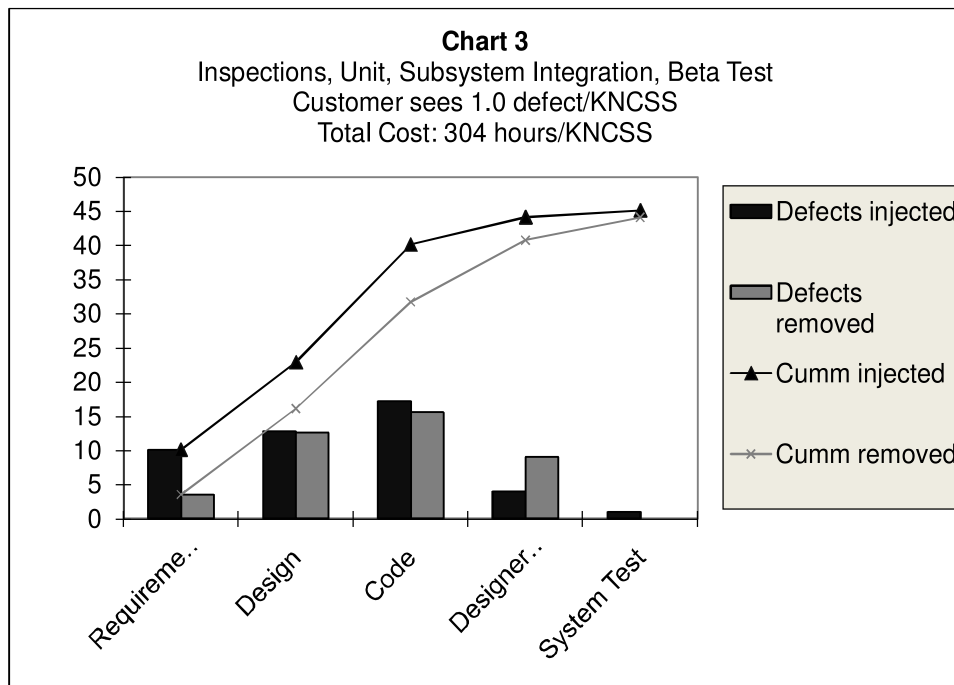




These two charts convinced management that we should inspect all requirements, specifications, and designs; and retrain every engineer on inspections. The data wasn't as compelling for code inspections, so we left it up to the teams to decide whether to use code inspections or informal code reviews. As part of this effort, we began tracking metrics on peer reviews and created formal checklists for our inspections. Many of the labs also started using plan-do-check-act each quarter on their inspection and review processes.

Then we looked at beefing up our testing on top of the inspections. What if we did formal unit testing, system integration, and beta testing? As you can see from Chart 3, our labor only was reduced by 15% (it's more costly to develop tests than to do inspections) but our defect rate decreased by a factor of 3. This was sufficient evidence to our management that they were willing to invest in both equipment and test development to improve our testing. We now needed to develop a test strategy to determine exactly where we wanted to improve.

You'll remember our goals were to reduce the defects found after check-in by 2X every 18 months and to find more defects between check-in and release. With this data, we actually decided to change our internal goals to emphasize finding defects by the development team – because it would reduce the total cost. The goal was changed to have the development team find 90% of their own defects (through inspections and test), then system test (and other test groups) would find 90% of the remaining defects, leaving only 1% to be found by customers.



Creating a Test Model

So after building a strategy to improve our testing based on the fact that it would save us costs and significantly improve customer quality, it was then time to figure out our test strategy – exactly what were we going to improve. Our starting point was to document the current strategy. We did this by creating a process model of our testing. Then using our defect analysis, we built a target process model – what the testing strategy should be.

Current Model

Tests of subsystem		Tests of entire product		
Module Test	Subsystem Test	IC Test	System Test	Solution Test
No criteria	>95% pass on functional test on 1 SPU)	>95% pass on functional test on a couple configurations	>99–99% pass on functional test (more varied and larger config)	0 defects running with key layered applications
		Run stress tests on small configuration	96 continuous hours of operation on stress tests	

Chart 4

Chart 4 represents the current model of our test strategy. Notice that it has two major subdivisions – testing we do on subsystems, and testing we do on the entire product. Development teams did module testing - the design engineer, knowing his design, did some testing to determine whether the module worked as he thought it should. Most engineers did this, but we had no clear expectations of what was required here. So in our future model we decided we should ask for unit testing, testing of less than 300 lines of code with branch flow goals.

We also had regression tests for every subsystem, which were primarily functional tests. The good news was that most of these tests were automated and run on a regular basis. We had also set criteria that they should always pass at least at a 95% rate. The bad news is that we really didn't know how much of the system these tests actually tested. (You'll remember that functional defects were the most common type of defect, so we knew that these had to be improved.)

We also had an Integration Cycle (IC) every 2 weeks where we put together the entire product. We would run most of the functional tests on these ICs and run some stress tests. The big difference between the IC test and the subsystem tests was that for subsystem testing we had the latest version of each subsystem and we tested on a larger set of configurations. For our stress tests, we measured what we called continuous hours of operation – how long the system would run before it hung or had a panic.

Finally, toward the end of a release, we would begin system testing. System testing would include many of the layered products as well as the UNIX operating system itself. We used the same functional regression tests, but in system test we had much larger variety of configurations, and in particular, this is where we would test on our large multimillion dollar computers. We tried to get to at least a 99% pass rate before we released the product. (Sometimes the tests would fail occasionally just because the test weren't written to handle all situations, so 100% pass wasn't practical.) Stress testing was also done on these larger configurations – with a goal to get two runs of 96 hours of continuous operation (CHO) in a row before we could release.

Finally, we had what we called solution test. Early on, this really consisted primarily of testing with HP layered applications and giving the product to our field engineers and Independent Software Vendors (ISVs) to test. You'll notice that our solution test didn't include any non-HP applications, nor did it reflect any typical customer configurations.

So we used our defect analysis to help us define our target model in Chart 5 (new tests are labeled in white). As part of our target model, we carefully defined the purpose of each kind of testing, its objectives and how we would measure success.

First, we looked at the big picture. Since over 60% of our defects were in the core product (NCKL), we actually added separate testing of this core to our test strategy. Since defects in the core affected everyone, we wanted especially to be sure that most of these defects were found before submittal. You'll see that we added three new test types to find defects quickly. Let's start at the top left and look at the reasons behind our chosen test strategy.

Module testing was formally renamed to unit test to indicate two things: a) we were expecting it to be done on smaller portions of code, not an entire subsystem. Typically, this would be 300 lines of code or less. b) We expected to achieve 100% functional and branch coverage by using a harness around this small set of code to be able to test all paths. Today, this is probably still our weakest area – largely because we didn't set up a mechanism to track that this was actually being done. It's also the case that we have so much pre-existing code (remember those 18 million lines of code), that it would be impractical to create unit tests on all that existing code.

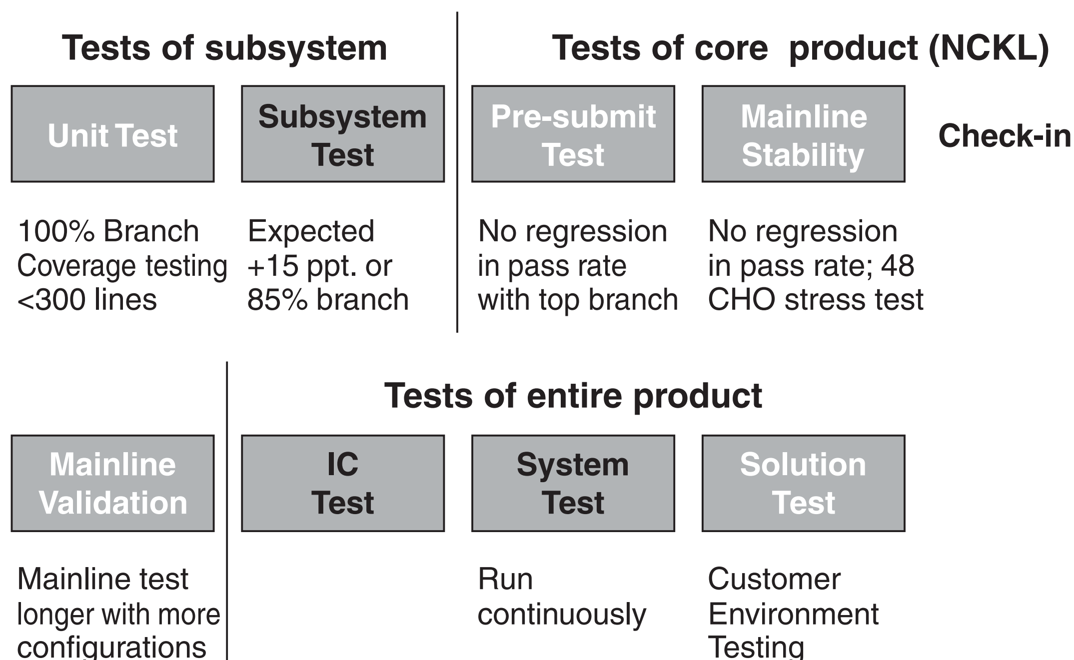


Chart 5

For subsystem testing, we added some criteria of goodness to our existing pass rate goal of 95%. We set goals to improve the functional and branch coverage of these functional tests. For new code, we expected 85% branch coverage and 100% function coverage. For existing code, we asked that for every major release, teams try to improve the coverage by 15 percentage points, from 50% to 65% coverage, for example.

Pre-submittal testing was a new test that was added and run by the developers. When they were ready to submit their code, they would run the functional tests against the top of branch for all the core modules. The goal was that there would be no regressions – all those tests that were passing before must continue to pass. This allowed us to slowly increase the pass rate of the entire system from 95% to 99.5%.

We then added a 48 hour test called mainline stability. Here we had a wider variety of configurations (including some of our high end SPUs) and ran the complete set of functional tests as well as 48 hours of stress test. If these tests failed, the check-in to core was rejected – we either backed out the submission or fixed it immediately.

After check-in we would continue to run our module testing in different configurations of networking and SPUs. Because differences in configuration accounted for 10% of our defects in our defect analysis, we wanted to increase the variety of configurations.

We continued to build the entire product every 2 weeks and run the same tests on it; this part of our test strategy remained relatively unchanged. Similarly our system tests didn't change much, except for one huge difference in the environment. Because our quality was higher, we actually could run these tests all the time, instead of waiting until the end of the release.

Our final change was to add solution test. We started running many more ISV applications on the system rather than just asking them to test. We also started building test environments that looked a lot more like our customers' environments. The test methodologies were enhanced to take on more of the customer style experiences: install/update, recover, data migration. The hardware configurations were expanded to fit our customer profiles, which typically consisted of 3 to 4 tier internet applications.

This has since been enhanced even further. We have representative customer application stacks for particular industries. We use trend-setting customers, those customers who tend to be the innovators in their industry and are stressing our systems the most. We've found that if we can get those applications stacks working properly, most customers in that industry will also benefit.

Results and Lessons for Others

In our first release with the new model, we reduced the customer-found defects by 6X, even though we added twice as many lines of code as the previous release. In that same release, we did achieve our goal that 90% of all defects were found by the developers. In our second release, we reached the goal of customers finding less than 1% of all defects. At the same time, we produced 30% more code/engineering month, reduced our backend testing time by 25% and reduced our current product engineering effort from 20% of our engineers to 13%.

The lessons for others are fairly clear:

1. Use formal inspections

   If you aren't using formal inspections, this is the place to start. Inspections are well proven in the software literature as key to finding defects earlier, when they are less expensive to fix. However, in order to convince management of their value, you may have to generate your own return on investment data similar to what we did.

2. Use defect analysis to pinpoint improvements for your quality process.

   Defect analysis builds its own support among the engineering teams because they decide how to fix the most frequent problems. Your defect analysis may also suggest changes in your test strategy. As you build the new strategy, each stage of testing should be well defined in terms of its purpose, objectives, and metrics.

The combination of these two techniques leads to both higher quality and productivity.

---

[i] "Management Commitment to Quality Requires Measures" by John Balza. PNSQC 2003

[ii] "Defect Analysis – a tool for improving software quality" by John Balza, PNSQC 2004

[iii] "Software Quality, Analysis and Guidelines for Success" by Capers Jones, 1997. Pp 138-140 (10,000 FPs size projects)

[iv]" Applied Software Management" by Capers Jones, 1991