Certification Does Not Equal Quality

by Paul Tsuda

If you're not developing software to control nuclear reactors, nor trying to sell in the commercial shrink-wrap market, this article is for you.

I work in a company that traditionally developed software for the cable television industry, stuff to do their subscriber management and billing. A saying used to circulate in the industry before convergence (combining telephone, cable and data communications into a one-stop service), 'It's Only Cable'. This usually meant it wasn't worth the effort to fix the last crop of minor bugs because failures would not be catastrophic nor life-threatening. The phrase also tacitly acknowledged that cable companies operated in a monopoly environment where it was the only game in town.

Well, convergence changed all that, along with competition from the satellite TV companies.

But getting competitive hasn't been easy, especially reining in the cowboy coding style of the company founders. In the mid-90's a new VP was brought in to tighten up the process, and within two years the company had its Process Certification proudly hanging on the wall.

But the problem then became one of enforcing the standard and getting people to actually think and act in a more structured way. You see, the founders worked endless hours to get a working product out the door. But when they left, with them went the visionary spark which made the process and the product come together. Without them it became easier to go home at a decent hour, or to spend the time at your desk working in a less focused and therefore less productive way.

As it turned out Certification did not produce higher quality code. The developers were able to go through the motions dictated by their new procedures but, without the motivation provided by their departed leaders, were unwilling to put in the extra effort that differentiates the Microsoft's from the also-rans. They had a formal QA department who performed inspections, walkthroughs and reviews but lacked both product experts to do final testing and visionary leaders who could keep the team working together during that final difficult push to get a polished product shipped.

Well, a new management team has been brought in to figure out why Certification failed. My hope is that they see what seems obvious to me: too much emphasis was placed on prevention activities (inspections) and not enough on appraisal (testing). Of course the obvious oft times goes unnoticed because we choose not to see the paths we know will be harder. I guess it all depends on how badly the new management wants to win the game.

And turning a blind eye toward an effective solution might not be the only reason real software quality is elusive. It also might have to do with the implicit thinking behind the software engineering paradigm itself.

Testing the Product, Not the Process

'It seems to us that every company already has a proper group to set standards, evaluate and train staff, and generally monitor and work to improve every phase of product development. That group is called Management. Management is the real quality assurance group in every company.'

- Kaner, Falk and Nguyen from *Testing Computer Software*

The first graphic projected on the overhead screen in a QA seminar shows a rising curved line representing the cost of bugs as the software project proceeds. It's displayed to demonstrate that bugs caught during the requirements phase are much less costly than bugs caught during the design phase, that the ones identified during design cost less than those caught during coding, and that coding defects cost less than test defects. Of course the big no-no are bugs reported after delivery. The cost of bugs reported by customers is huge, and many a creative graphic artist has depicted this fact by making them appear larger in the later phases of development (a good example can be found on page 18 of the book *Software Testing*, by Ron Patton).

I have a problem with this graphic. I think it's original intent was to dramatize the necessity of front-loading the development effort, i.e., not skimping when it comes to doing requirements analysis or program design. The problem is that now it's used to downplay the importance of system testing and the subject matter expertise needed to do it right. It's also encouraged a QA culture that emphasizes the role of tester as process-expert rather than product-expert. How many times have you heard a QA person say that 'you can't test quality into a product, it has to be built in'. To me, they are saying the narrow role of 'tester' ignores the more important role of identifying bugs in the development process itself.

Getting back to the bug-cost graphic, I look at it in the following critical way: the small bug depicted in the requirements phase is the cost of finding a *requirements* bug in this phase; the larger one in the design phase is the cost of this same *requirements* bug if found during this phase; and the largest one looming above the delivery phase is the cost of this same *requirements* bug if found during this later phase. **The fact that these are all the same bug gets lost in the chart**. And the subtle message slipped into your mind is, "it's terrible to allow bugs to live this late in the development process." (Cleanroom fantasies of software development may begin to germinate in one's fertile brain).

But if you look at the graph in this *new* way the defects found in the final testing phase are not so big and scary. Instead of seeing them as monsters that should have been caught earlier in the development cycle, they become normal sized because few will have been created during the requirements phase. This more critical way of 'seeing' the chart might even make finding bugs in the system test phase seem as natural and important as finding bugs during requirements or design.

I hope my point is coming across: testing during the final stages of development is just as important as creating a solid requirements or design document. As a matter of fact, common sense tells you that testing is one of the most useful tools to ensure a quality product. Why do you think we require doctors and lawyers pass tests before getting their license to practice? And why do you think professors test their students several times during the semester, and base final grades on the outcome of these tests? I think it's because we all know that 'the test' motivates people to do their best. The end result is a doctor we can trust and, when applied to the software engineering discipline, a program we can rely on.

So besides turning a blind eye and hidden assumptions what else could be a root cause of the elusiveness of software quality? One explanation might be the inevitable result of both human nature and the nature of problem solving in general.

Development's Natural Flow

"*The complexity of practice has always dwarfed the simplicity of theory*"
- Robert N. Britcher from *The Limits of Software*, 1999

In larger programming shops that use software engineering methodology (any waterfall-like approach) to create non-safety-critical applications (no air traffic control systems), work naturally flows away from *appraisal* activities like system testing and toward *prevention* activities like requirements, design and code reviews, for the same reason that water flows around an obstacle: it's easier.

In the software quality assurance industry there is a formal process called V&V (Verification and Validation). *Verification* has to do with prevention. It's when a requirements document is scrutinized to verify the development team is creating what the customer wants; or when a design specification is verified to make sure the proposed product satisfies the stated requirements; or when a programmer's code is reviewed for fidelity to the design objectives. It's easier for developers to focus on these documents because they are basically 'theoretical' in nature. No real product (program) exists. At this point there is no right or wrong. The customer will never see this stuff!

Now let's talk about the other 'V', validation. *Validation* has to do with appraisal, testing. It's when the programmer actually unit tests a module, or when a QA analyst does system testing. At this point a concrete object exists, the program, and it either works or it doesn't. It's usually right or wrong, valid or invalid. This is the hard part, the nitty-gritty detail work. The programs are tested, and if they don't work, they are fixed and retested. This is repeated until the tester says that the program is 'right' (or at least right enough to release to the customer). Most people, like flowing water, will try to get around this kind of work because it's really 'work'. The customer actually sees this stuff, and if it's wrong they won't be happy.

So this is my point: if you're leading a development effort your job is to keep the group's activity focused on the tasks that make a difference in the quality of the *delivered* product. Since your task is both prevention and appraisal you have to keep the effort balanced on both kinds of activities. Of course, your life will be a lot easier if you concentrate more on prevention, but you can't let that happen. I know it's tempting—when it gets down to the system testing phase almost everyone just wants to get the product validated and out the door so they can start working on something new and more interesting. But the only time 'easier' makes sense is if you don't have any competitors. If this is the case, it's much more practical to let customers do the hard work of catching those pesky little bugs. But not everyone has this luxury. Do you?