It's The End Of QA As We Know It, and I Feel Fine

Reevaluating our product development model

What's it cost?

There is a commonly described association between the cost of a bug and when the bug is found (and presumably fixed). It goes something like this— if a bug found during the development phase costs $1 to fix, then, if found in Unit Test, it costs $10; if found in System Test, $100; and if found in the field, $1000. The exact numbers are arbitrary but the order of magnitude effect is not. It's pretty easy to imagine this effect—if, while writing a line of code, a bug is found and fixed, the impact is usually on one person, and may take only a few minutes to resolve. If the defect makes it to the next level of scrutiny, the tester discovering the problem now must enter a bug report, there must be some tracking mechanism, and after a repair is made, the fix must be verified. This process sounds expensive. At the System Test level, bug fixes may require significant amounts of regression testing to ensure no other area of code was affected by the fix. This sounds very expensive. Once a bug makes it to the field, it gains a new level of impact—it can affect any number of your customers. That one bug can hit you over and over. Not only does this sound extremely expensive, it sounds like it could hurt—a lot!

A few caveats—first, it is very difficult to measure the cost associated with a bug found in the field. We can only quantify the actual internal costs associated with the problem: support costs, field engineering time and travel, and so on. The one, and I suspect, by far the largest cost, we can never really know—customer reaction. Does the customer expect some future discounted product or service as compensation? Do they return the product for a refund? Does the damage caused generate legal action against your company? Do they simply badmouth your product and/or company to others (potential customers)? The answer to most of these questions is simply that you don't know. Rather than trying to find better answers, we should consider how to avoid these questions altogether.

Second, there are business decisions to be made here. Even if we can quantify a cost associated with a particular bug making it to the field, there may be legitimate decisions to not fixing the problem. From a QA perspective this is heresy, but from a business perspective, delaying a project to make that fix may actually cost a thousand times the fix cost in lost revenue—it's that whole time to market thing, where the sooner you ship, the more money you make.

Rethinking the Problem

So how do we avoid these expensive customer questions? Simple. Find the bugs earlier. Problem solved. Next topic.

OK, so finding bugs earlier in the process is the real issue we want to look at here, but there are problems associated with this ostensibly simple approach. The "traditional" 5 phase development strategy—Marketing Requirements, Architecture, Low-level Design, Implementation, and Validation, puts the test process at the end (validation); after all the decisions have been made—think of it as the "Inspector 12 Model." This model sets the Quality Assurance organization as the "safety net" for quality. If something bad made it through all the other phases, it is hopefully caught by test. This often saves the day, but more often appears to be an impediment to launching the product. "Why aren't we ready to ship?" asks some top-level manager. "We've finished the product and now we're all waiting on QA to look at it," as if QA is simply some bureaucratic organization whose sole responsibility is to smack the product with a rubber stamp after waiting some predetermined period of time. So everyone puts the pressure on QA to "assure quality." Bad idea. Given the realities of time to market, in this model, QA can't assure anything. All it can do is point out problems that assure "non-quality." Better to rename the organization Non-quality Assurance. Better yet, maybe they should be involved earlier in the process.

Some "progressive" development organizations have done just that—move the QA effort earlier in the development process, but common implementations of this scheme fall well short of the solution. In this model, the QA process is part of a 'looping' between the Implementation and Validation phases, where interim product builds normally seen in any development program are subjected to the 'external' eyes of the QA team. Problem areas normally found much later in the process are flushed out, and fed back to the development team. The development team implements changes needed to overcome these problems and feeds the resulting product back to QA. After some number of loops through this process (more on this in a minute), the product is deemed acceptable, and is sent out the door. Everybody praises the collaborative efforts of both Development and QA and the world is a happy place—except that the bugs could have been found earlier (read as "less expensively").

This looping process described above is an interesting problem in itself. How many loops is the right number? 3? 7? 25? I've seen real-world projects where each of those numbers was used. Is there even a magic number in the first place, or does some level of product quality that must be met determine the number of loop iterations? Are we looking for no more than 6 bugs before the loop can end? How about zero? Again, I've seen both of these numbers established as official project standards. (The zero figure seemed a tad unrealistic in view of the fact that, while no software can ever be bug-free, this product had almost 250,000 lines of code, which was sure to have at least some small number of defects. If that product ever made it out of test, it was surely because program management relaxed that zero standard or told the test organization to 'stop finding bugs'). So, we have an idea of the flaws in these two examples. Understand that this doesn't mean to suggest that we eliminate criteria altogether, only that we give some credence to what those criteria really mean in terms of product quality. Also understand that, while some kind of loop process is inevitable, finding bugs at this stage in the development process described above is still very expensive. We need to identify problems even earlier.

How about putting the test organization, the one you claim is responsible for "quality assurance," at a point where it can actually impact quality—and even go so far as to assure it? You're probably wondering where that point is. Simple, it is at every place where product decisions are made, without exception. QA in the Validation phase (of course), in the Implementation phase, during Low-level Design, during product Architecture, and while Marketing Requirements are being drafted. Note: I'm not suggesting that each of the separate teams involved in these phases build some kind of quality assurance team or quality analysis into their own processes. I've assumed they already do something like this (and having been involved in oversight of such things, it is a very bold assumption). What I'm promoting here is an integration of QA Engineering staff, management, methodologies, processes, and procedures into each of these phases.

In this (albeit radical) model, QA involves itself from the conceptual beginnings through the project phases, and into product release. Taking this one step further, it is arguable that QA should be engaged in the support and product maturation cycles as well (but I'll save that argument until after I win the first one).

In the computer industry, time to market is most important. It doesn't really matter how wonderful your product concept is, or how high its quality, if your competitor beats you to market, you're going to lose the ever-critical market share. With this in mind, why would anybody want to add the time and effort of having QA as part of each phase? Because, in doing so, you'll reduce the Validation phase of the product cycle—so much so that, if properly devised, you can actually shorten the entire development process over the conventional "test it last" model. And, where you don't actually shorten the schedule, you'll still have a better product—a product that is quality assured. And that product will cost you less to produce.

To successfully implement this bold plan, a major paradigm shift must occur within the larger organization. Since these don't usually happen spontaneously, you're going to need buy-in from upper management to 'legislate' the process (you will need everybody's management in on this, not just QA), then diligence in implementing and enforcing the change. It sounds like I'm describing controversial criminal legislation with words like 'legislate' and 'enforcing,' but we would be lucky if it were something as simple as that. At least there you get to use handcuffs and billy clubs. In this case, you'll likely be fighting an established mindset of "Us v. Them" (where "them" is basically everybody you're trying to work with).

QA in the Marketing World

Marketing people are very good at what they do. Just ask them. After you get their answer, tell them you're there to do some of their work for them… so that they can attend to more important matters. It's the second phrase that will get their attention. Just don't let them assume you mean to do their dry cleaning.

Every person I've ever met with the word "marketing" in his or her title was, for some inexplicable reason, excessively busy—either real or imagined, they were busy nonetheless. These are the people who are always 5 minutes late to a meeting and leave 5 minutes before the meeting ends. They are the ones who have their notebook planted firmly in front of them, furiously typing, while simultaneously scribing gibberish into their PDAs. How they are able to read emails, watch (and sometimes even conduct) the overhead presentation, and participate in discussions is beyond my comprehension. That is, apparently, because they are very good at what they do. Of the many I have met, there stand out those few who are exceptional at what they do. Take the list of tasks above, add a cell phone conversation, in the meeting room, and which possibly involves making a detail shop appointment for their new fire engine red exoti-car or securing reservations for their next (of many) vacation to their favorite exoti-island. That is because these marketers are exceptional at what they do. While all of this seems like marketer-bashing (I concede that some of it actually is), its real purpose is to show that the marketing group only has time for genuine marketing issues, and that is alright—that's what they do best. They do not have time to learn, understand, and implement quality assurance methodologies, processes, and policies.

The QA organization's task now is to help them with the things that they are not suited for, but have traditionally been responsible for—developing realistic product objectives. When Marketing suggests that the new product X needs to sprout wings and fly, the QA team needs to be there to explain the effort required to test this feature so some kind of initial cost-benefit analysis can be made to determine how important it really is. The test organization must determine what level of flight constitutes quality so that the product can be architected to meet that expectation. In the traditional models, QA doesn't even know the program exists until after the specifications are complete, when it is too late to enlist their expertise.

Quality Architecture

Several programs in which I have been involved used a model of architectural dialog that went basically as follows—the marketing team presented its (wholly unrealistic) product requirements. The architecture group responded with their ("you've got to be kidding me") straw man overview. Marketing responded with a more realistic requirement set (which now, at least, doesn't violate known physical laws). The engineers in Architecture responded with more details. And this process continued (ad nauseum) until a design specification was hashed out. (Some refer to this as "marchitecture." A word that, by its sound alone should conjure up images of unholy deeds being done.) The problem with this scenario (actually there are several, but let's focus on the one related to quality assurance) is that there is no connection to quality. Marketing says they want certain features and, after much commotion, the engineering team has defined an architectural overview of the project that is to contain these features. Where are the quality benchmarks? Where is the quality assessment mechanism? And what will constitute acceptable quality?

To answer these questions, the QA organization must be integrated into this product specification exchange. Include a QA response document that addresses design issues affecting quality. Present quality oriented solutions to architectural bottlenecks. Establish the bar for measuring successful quality attainment for the feature set. Make these measures part of the project feature specification document itself.

Low-level Design

Now that the feature specification is written, it's time to detail functionality—sprouting wings can be done by this and that, but just how are we actually going to get this thing to fly? It's this functionality specification that will detail the actual structure of the product.

The QA team has the most significant opportunity for impact in this phase—though it's not until later in the process (as was described in the "progressive" approach above) that QA has a chance to flush out actual code bugs. Currently, many organizations have test automation processes that create some level of non-interactive testing. And we have third-party tools and test scripts to take care of the more mundane aspects of testing that generate repeatable results. Imagine extending this automation concept to the next level—to a level that is integral to the project code itself. What if the program could test itself? Or at least assist the QA team in testing. While the topic of self-documenting code is hardly new, the concept of self-testing software seems oddly foreign. Coders are aware at the microscopic level that testing their software is important. They stub out code modules with end points that allow them to exercise the code without the full program in place. They do this, ostensibly, to make sure the section of code they are working with actually works. But most of the developers I have spoken with (and most testers and program managers as well) look stunned when I suggest that we build a program to test itself. If that doesn't floor them, I usually throw them my second bit of radicalism by suggesting that the program should also self-analyze, diagnose, and repair. That one usually wipes their faces into a completely blank stare. But since we're still looking for some buy-in, lets not go overboard, and instead stick to the self-testing

software idea (for now!). Developers may not believe it, but they already are writing self-testing code (or, in the following examples they should be).

In a simple example, where an application requests a user's United States Social Security number, there are only a few response modes for the end user—1) no response at all, 2) an incorrect response, or 3) the correct response. So, how does the program test itself? By validating the input information. First, Social Security numbers always follow the format xxx-xx-xxxx, where each place must be an integer from 0-9. (Other rules apply but are not relevant in this simple example. They may be relevant in the real world, however. Make sure you understand the rules.). This means that the code should expect only numbers between 0 and 9, and should only expect them in the available field places (xxx-xx-xxxx). Too many numbers and the program should throw an exception message. Letters or other non-numeric characters, throw an exception. No entry at all, if the user tries to proceed to another screen, another exception (or at least a confirmation since some users may not have a Social Security number. Make sure you understand the requirements.). These software checks are simple, resolve all three of the user response modes from above, and the software is essentially testing itself. If the right set of constraints is placed on the input field, it is very unlikely that this area of the code will exhibit problems. So all that must be tested in this simple example is whether the actual input is constrained. Once we determine that, the rest of the code can test itself. We're done. Ship it.

OK, so this is a very simple example, of a very simple piece of code, in a rather "obvious" bit of self-testing, that is often already done. But why do we stop here? Why does this constitute the end-all of self-testing code? What about adding this kind of code testing throughout the project? On module outputs, determine what would be an acceptable value and, if what is passed doesn't meet acceptability, throw an exception. As these check points permeate the code, the entire product begins to test itself. All that must be manually considered is checking the 'check code' itself. Code reviews can increase our confidence that these self-tests are working, and a handful of spot checks will make us warm and fuzzy.

While this may sound like a gross simplification, and that the actual implementation of this new 'supercode' will likely overwhelm the development staff, consider this—James Whittaker, Associate Professor of Computer Science at Florida Institute of Technology describes software as only doing 4 things-- software receives input, makes calculations, stores data, and sends output. Everything else is just some form of these functions.

So, in the Low-level Design process, QA must insist on testing these key areas. And they must insist on automating these tests—by automating them within the operation of the code itself. In doing these things, the amount of traditional testing is substantially reduced.

Implementation and Validation

Now that the code is doing most of the work of testing itself, all that is left is a greatly-reduced form of the looping process described earlier; the one where QA and Development work through the product to identify and fix the remaining defects. A final "assurance of quality" validation by the QA organization culminates this phase, where the product can then be released to the customer.

Creating a cultural shift of this type is a significant challenge, and implementation must be carefully planned. The end result, however, can significantly improve product quality and decrease development time—both of which are in high demand.

Abstract & Author Information

Excerpted from a book-in-progress, this article explores the concept of integrating quality into all areas of the product development cycle and some benefits to this model.

Kenneth Hass has been involved in development, test and quality assurance for 15 years. His efforts have included computer hardware, software, and consumer electronics products. His current focus is on improving products by integrating the QA process through the entire product lifecycle. He can be reached at labboypro@hotmail.com.