Not-Defect: The Mature Discipline of Testing

Taking a deliberate, measured approach to software development and testing.

Many years ago, Glenford Myers asserted that the activity of testing software systems was “out of vogue” and complained of the scarcity of literature on the subject [3]. While the subject of testing has generated a great many books since Glen Myers’ lonely offering in 1979, testing is still perhaps not accorded its due. An informal search conducted on Amazon.com under “software testing” produced a healthy 276 results. But a similar search under “computer programming” maxed out at approximately 32,000 hits, and a search of Java alone produced nearly 3,000 book references. Not that sheer numbers of different books indicates a maturity in a discipline. Indeed, it may actually indicate the opposite. It may be a symptom of the variety of different ways in which different people think systems should be built. But when an obscure language like Forth apparently has one-quarter of the number of books that are written for a whole discipline like testing, further investigation is required.

Two Kinds of Work

In this column I have been exploring the idea that, despite common practice, developing software is not a product manufacturing activity. Rather, it is a knowledge acquisition activity. The thing that kills us in software is never what we know—it is what we don’t know that causes the damage. And it is fairly obvious that the problems we experience in software (defects, project overruns, project failures, and so forth) are primarily a function of what we don’t know. That said, and while resolving our ignorance is paramount, we do have to expend some effort to factor our extant knowledge.

So there are two kinds of work we do in creating software systems:

• The application of what we know; and
• The discovery of what we don’t know (followed, of course, by its application).

These are quite different activities. We experience significant problems with software process definition initiatives when we attempt a one-size-fits-all approach. We sometimes try to use the same process for application of what is known to the activity of discovering of what is unknown, or vice versa.
It’s easy to see that factoring what we already know into an executable form is usually not too difficult. Problems surface when we attempt to do this only to find out that we really don’t know what it is we should be factoring or we create an executable only to discover when it doesn’t work that we really don’t know what we thought we did know. At this point, we are forced to stop constructing what we thought it would be and start learning what it should be.

Given a 100% complete specification and perfectly defined design, we can create a flawless system just about as fast as our fingers can type. But that has never been the issue in developing systems. Personally, after 33 years in software, I have never received anything remotely like a “complete” specification, and I seriously doubt the existence of such a thing.

Since the application of what we know is relatively effortless, and the completeness of what we are given when we are asked to build systems is marginal, obviously it is the “discovery of what we don’t know” that causes problems for us.

**Competence Versus Ignorance**

Our problems start appearing when we (try to) build systems. Given the almost universal product building mind-set in the business of software, we are considered “successful” when we write a complete specification, we create a feasible design, or we build something that works. This is quite reasonable—our job is to build something that works. The problem is how we go about it. If there are things I know and things I don’t know, I can most rapidly create a “complete” specification if I write something that consists entirely of what I already know. Ditto design and ditto code. While we could certainly argue that the purpose of requirements gathering is to expose those things of which we are unaware, in practice, it often turns quickly into an activity of fervently cataloging what we already know or what we guess is correct in order to make the specification milestone rather than pursuing what we don’t know, which is too, well, slow.

This preference to work on what we know is a very natural human tendency. All through software development, we are presented with choices in our ways of working. One choice is to exhibit our apparent competence by quickly creating a polished and complete artifact for which we already possess the knowledge and experience (which is how it quickly ends up “polished and complete”). The other choice is to work on something about which we haven’t a clue and in doing so we will often make mistakes and display apparent incompetence. Now I don’t know about you, but given a straight choice between working on something that I know very well or working on something about which I haven’t a clue, I’m going to choose the one that makes me look good. In fact, I confess I have a set of personal hobbies that I have cultivated for years about which I feel comfortably competent. This is why I keep them around, of course; I think they make me look good. I assiduously avoid doing anything that makes me look like a klutz.

The problem is by doing this I don’t tend to learn anything new. The same is true of software development. And since software development is primarily a learn-
ing activity that just happens to deposit the results of the learning in a place that executes, this is a problem.

Building Things or Learning Things
Almost all stages of software development focus on building things. Even the requirements gathering stages quickly segue into requirements specification writing stages. This is not a fault of the requirements stage per se, it is simply a subset of the whole product production paradigm. But it is quite dangerous nevertheless.

If we look closely at almost all of the development activities, there is this strong undercurrent of “build it”—finish the plan, write the specification, complete the design, churn out the code, make the milestone. The conviction is that the artifact—the plan, specification, design, code—is the thing. It is not. The thing is the knowledge that goes into the artifact; it is not the artifact itself. It is sometimes a subtle point, but I feel strongly it is the point. We do not manage software development as a learning activity, we manage it as a production activity. We barely manage the artifact and we hardly manage the real product at all. A classic example of this is our use of Configuration Management (CM) systems. While our management of configuration items has become much better than it used to be, it is still pretty coarse. If we instruct our CM system to give us version 1.1 of the specification it will dutifully do so. But what is in that version of the specification? Well, you will have to read it to find out. The CM system does not have visibility into the contents of the spec. It is not granular enough. Our CM system is not managing the knowledge product; it is managing the bucket into which we put the knowledge product.

Our knowledge-connected model-based engineering approaches are getting us closer to better management of the contents of the bucket, but they are still impeded by the fact that organizations are predicated on the production and shipment of products rather than the discovery and management of knowledge.

And all phases of development, even phases in the more agile of life-cycle models, tend to operate on this principle. Except testing.

An Epiphany of Knowledge
Testing as a directed activity is the only part of software development that has truly adopted the mind-set that our job is not to catalog what we know, it is to discover what we don’t know. Most testing organizations understand and appreciate that their job is to intentionally find “defects.” But what exactly are defects?

Simply put, a defect is a lack of knowledge made manifest. A defect is something we don’t know. It might be a requirements something or a code something or a platform interaction something, but it is a something we have not (yet) learned. The exposure of a bug is what I call “an epiphany of knowledge.” A bug is a system’s way of telling you that you don’t know something. But discovering what we don’t know is our job. And only testing has made this its prime directive.

Two Kinds of Testing
We do two basic kinds of testing: the “proof” of what we know and the discovery of what we don’t know. These correspond to resolving Zeroth Order Ignorance (0OI), which is knowledge (or lack of ignorance), and Second Order Ignorance (2OI), which is what we don’t know we don’t know [1]. Note we would never test for First Order Ignorance (1OI) manifest lack of knowledge (what we do know we don’t know). This would be tantamount to saying “I know we have a table overflow problem right here in this program and I’m going to run a test to prove it.” If we did know we had a table overflow problem, we’d fix it first and then run a 0OI test to show it was fixed. Since our clean “prove it works” tests are usually fairly straightforward, we spend most of our testing time and effort look-
ing for the 2OI things we don’t know we don’t know. This presents a problem. It is very hard for scientists to develop experiments for something for which they are not looking. And it is difficult for testers to create tests to expose something they don’t have a clue about.

Two Stages of Testing
There are always two stages in testing: expose a bug; and fix a bug. These correspond to the two cycles of 2OI \(\Rightarrow\) 1OI (convert what you don’t know you don’t know into what you do know you don’t know) and 1OI \(\Rightarrow\) 0OI (convert what you do know you don’t know into what you do know you do know). While the second stage of tracking down the effective solution to a bug can often be quite difficult, it is usually easier than the first stage. For sure, we cannot fix a bug if we continue not to know it is there, so the 2OI \(\Rightarrow\) 1OI tends to be the more critical step.

The Maturity of Testing
Testing is the only discipline in software development that has really stepped up to the plate and acknowledges what the job really is—to find out what we don’t know we don’t know. Interestingly, we could argue that testing is also the area where most truly executable software is used in the development activity. The earlier specification, design, and code stages typically use word processors or similar applications for the bulk of their computer support. Such tools are closer to a form of electronic book (you can write in them and then you can read what you wrote) than they are to executing software, since often they don’t actually do anything. Good test software, on the other hand, doesn’t just describe the knowledge of testing a system, it executes the knowledge of testing a system: there’s a big difference between the two.

Not-Defect
Given that our job is to find out something we didn’t know, does finding this out constitute a “defect”? I could make a good case that the answer is no. Apart from the semantic (defect = defective), is exposing a defect a bad thing? The discipline of testing has grown to the point where we now realize that exposing a “defect” is a good thing—in fact, the process of testing it is the thing. But if discovering a defect in testing is the only or the most efficient way we can find out that what we thought was right was in fact wrong, or what we thought was complete was deficient, this is not defective at all. Strictly speaking, a defect is something we should have gotten correct but didn’t for some reason. A defect is some evidence of negligence or sloppiness, not merely the exposure of something we didn’t know. Heck, all of our activities should be the exposure of something we didn’t know. It just so happens only testing completely realizes this is true and then acts like it is true.

The Art of Software Testing
In The Art of Software Testing, Glenford Myers asserts “…the most important considerations in software testing are issues of economics and human psychology” [3]. He was right. Of course, the most important considerations of any part of the business of software are issues of economics and human psychology. But we get the point. Until the rest of the development phases catch up, testers can compensate themselves for the relative lack of attention to their craft by reflecting on their relative maturity with respect to all those designers and coders.

Hey, maybe someone will write a book about that.

References

Phillip G. Armour (armour@corvusintl.com) is a senior consultant at Corvus International Inc., Deer Park, IL, and a research director in the Center for Software Development Innovation at Number Six Software, Arlington, VA.