The Ongoing Revolution in Software Testing

Cem Kaner, J.D, Ph.D.

Software Test & Performance Conference, December 8, 2004

This article is copyright (c) Cem Kaner, all rights reserved. Preparation of this article was partially supported by grant EIA-0113539 ITR/SY+PE: "Improving the Education of Software Testers." Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Twenty-one years ago, I started writing *Testing Computer Software* (Kaner, 1988), a book whose second edition (Kaner, Falk, & Nguyen, 1993, 1999) outsold every other book in the field. It didn't seem destined for the mainstream back then.

I opened with these words:

> Some books say that if our projects are not "properly" controlled, if our written specifications are not always complete and up to date, if our code is not properly organized according to whatever methodology is fashionable, then, well, they should be. These books talk about testing when everyone else plays "by the rules."
>
> ***This book is about doing testing when your coworkers don't, won't and don't have to follow the rules.***
>
> Consumer software projects are often characterized by a budget that is too small, a staff that is too small, a deadline that is too soon and which can't be postponed for as long as it should be, and by a shared vision and a shared commitment among the developers.
>
> The quality of a great product lies in the hands of the individuals designing, programming, testing, and documenting it, each of whom counts. Standards, specifications, committees, and change controls will not assure quality, nor do software houses rely on them to play that role. It is the commitment of the individuals to excellence, their mastery of the tools of their crafts, and their ability to work together that makes the product, not the rules.
>
> The development team has a vision of what they want to create, a commitment to getting as close to the dream as they can, and a recognition that they'll have to flesh out details by trial and error. By the time they've worked out the final details of one aspect of the product, they have a working version that one or two key people understand fully. That version is the "specification." It is not engraved in stone: it will be reviewed and polished later, to achieve consistency with other parts of the system. Much of the polishing may be at your suggestion, as the tester.
>
> In the real world, product changes are made late in development. When you're developing a product for public sale, your customers—your *potential* customers—did not agree to any specification. If a competitor creates something more appealing, you have to respond quickly.
>
> Desirable features are frequently dropped because there is no time to implement them. Code rewrites are also set aside, even if they're needed to bring a fragile first working version to a level the programmer considers professional. A conscientious programmer might take the initiative and do the job anyway, "on the sly." Late in the project, he may drop a significant change into the package without warning. These efforts go beyond the call of duty and the 40-hour week. They may improve the product tremendously or destabilize it badly, probably both for a time. Whatever the result, people take personal initiative to improve the product.
>
> There are always late changes. The goal is to cope with them—they are needed—as painlessly as possible. The goal is not to create a bureaucracy to inhibit them. If you can't keep up with them as a tester, you have a problem. But the solution won't come from complaining about them or in trying to stop them.

In writing the book, I intended to strip away many of the excuses that people use to justify bad testing, excuses like these:

- Excuse: You can't do good testing without a specification.
- Excuse: You can't do good testing without reviewing the code.
- Excuse: You can't do good testing if the programmers keep adding functionality while you test.
- Excuse: You can't do good testing if you get along too well with the programmers.

What I said in the book, and what I expected of (and saw in) testers who worked for me, was that people can do excellent testing under challenging circumstances.

This challenged some traditional ideas, and the book was sharply criticized for it. But I accepted and advocated other traditional ideas and pulled my punches on other practices (such as invalid metrics) and industry standards (such as IEEE 829 on test documentation) that hadn't worked for me, but I wasn't yet ready to condemn.

Since then, many of those ideas and practices and standards have become enshrined as *best practices*—which is marketingspeak for ideas that consultants find salable. Others have propogated into DoD standards, ISO standards, FDA expectations, and the overpriced practices of consulting firms that feed off the Sarbanes-Oxley fears of the Fortune 500.

I've been reappraising the assumptions about testing and development that I accepted or left alone back then. Several of us have been trying to write a third edition of *Testing Computer Software* for over seven years, but it keeps not working out. The book needs too many changes, not just in the detail but in the underlying assumptions. We made a start at reappraising the field's assumptions in *Lessons Learned* (Kaner, Bach, & Pettichord, 2001).

In the summer of 1999 (while Sili Valley was still booming), I decided to go back to school, partially because it was becoming clear that the changes in development were going to require a new generation of better educated test architects, and partially so I could step back and gain perspective on a field in evolution.

This paper for STPCon is my next crack at a paper that reappraises the assumptions about testing and development that I accepted or left alone in 1983, and perhaps the start of a new map for TCS 3.0. Some of the assumptions were probably always wrong. Others work under some circumstances, but not others—and in 21 years evolution of software development, circumstances have changed a lot. Others, I think, retain lasting value.

Let's take a look. Here are some of the assertions that I think are most widely made:

The Role of Testers

- Testers are THE advocates of quality on a project.
- Test groups should evolve into quality assurance groups.
- The test group should have the power to block release if product quality is too low.
- Testers and programmers have conflicting interests.
- The test group should work independently of the programmers.
- Testers should push their project teams to follow "appropriately professional development models," like the Waterfall, that require people to THINK before they act.

- Testers should base test cases on documented characteristics of the program. If the software documentation is inadequate for this, testers should assert a quality control function and demand better specification and requirements documentation.
- The primary reason to test is to find bugs.
- The primary reason to test is to prove the program works correctly.

Test Planning and Documentation

- Testers should specify the expected result of every test, in advance.
- Exploratory testing, if done at all, should be done only by experts.
- There should be at least one thoroughly documented test for every requirement item or specification item.
- Testers should design most or all tests early in development.
- Testers should design all tests for reuse as regression tests.
- Testers should document manual tests in great procedural detail so that they can be handed down to less experienced or less skilled testers.
- Testers should document each test case individually, ideally storing them in a test case management system that describes the preconditions, procedural details, postconditions, and basis (such as trace to requirements) of each individual test.

The Practice of Testing

- Testers should assume that the programmers did a light job of testing and so should extensively cover the basics (such as boundary cases for every field).
- Testers should develop automated tests at the user interface level.
- Testers should design tests without knowledge of the underlying code.
- Testers should design the build verification tests, even the ones to be run by programmers.
- Most testers don't need to be programmers, because they only have to specify how the external program will behave in their tests. Testing tools will allow business experts to develop automated tests without having to program.
- The pool of tests should cover every line and branch in the program, or perhaps every basis path.
- All failures must be reported into a bug tracking system and carefully counted.
- We can measure the individual effectiveness of software testers by counting how many bugs they find, perhaps with an adjustment for bug severity.
- We can tell how close we are to release of the product by examining the bug curve that shows bug finds per week.

The Role of Testers

Projects differ. Organizations differ. Products and their clients and their risks differ. Rather than a specific answer good for all situations, the role of testers (and the testing they do) has to depend on the context in which they operate. (Kaner et al., 2001)

The primary reason to test is to find bugs?

This is widely accepted wisdom. Even Glen Myers said it, in the first book published about software testing (Myers, 1979) and we promoted the same position (Kaner et al., 1999).

Actually, though, we test for many different reasons. Here are some examples (Kaner, 2003b):

- ***Find defects.*** This is the classic objective of testing. A test is run in order to trigger failures that expose defects. Generally, we look for defects in all interesting parts of the product.
- ***Maximize bug count.*** The distinction between this and "find defects" is that total number of bugs is more important than coverage. We might focus narrowly, on only a few high-risk features, if this is the way to find the most bugs in the time available.
- ***Block premature product releases.*** This tester stops premature shipment by finding bugs so serious that no one would ship the product until they are fixed. For every release-decision meeting, the tester's goal is to have new showstopper bugs.
- ***Help managers make ship / no-ship decisions.*** Managers are typically concerned with risk in the field. They want to know about coverage (maybe not the simplistic code coverage statistics, but some indicators of how much of the product has been addressed and how much is left), and how important the known problems are. Problems that appear significant on paper but will not lead to customer dissatisfaction are probably not relevant to the ship decision.
- ***Minimize technical support costs.*** Working in conjunction with a technical support or help desk group, the test team identifies the issues that lead to calls for support. These are often peripherally related to the product under test--for example, getting the product to work with a specific printer or to import data successfully from a third party database might prevent more calls than a low-frequency, data-corrupting crash.
- ***Assess conformance to specification.*** Any claim made in the specification is checked. Program characteristics not addressed in the specification are not (as part of *this* objective) checked.
- ***Conform to regulations.*** If a regulation specifies a certain type of coverage (such as, at least one test for every claim made about the product), the test group creates the appropriate tests. If the regulation specifies a style for the specifications or other documentation, the test group probably checks the style. In general, the test group is focusing on anything covered by regulation and (in the context of *this objective*) nothing that is not covered by regulation.
- ***Minimize safety-related lawsuit risk.*** Any error that could lead to an accident or injury is of primary interest. Errors that lead to loss of time or data or corrupt data, but that don't carry a risk of injury or damage to physical things are out of scope.

- ***Find safe scenarios for use of the product (find ways to get it to work, in spite of the bugs).*** Sometimes, all that you're looking for is one way to do a task that will consistently work--one set of instructions that someone else can follow that will reliably deliver the benefit they are supposed to lead to. In this case, the tester is not looking for bugs. He is trying out, empirically refining and documenting, a way to do a task.
- ***Assess quality.*** This is a tricky objective because quality is multi-dimensional. The nature of quality depends on the nature of the product. For example, a computer game that is rock solid but not entertaining is a lousy game. To *assess* quality -- *to measure and report back* on the level of quality -- you probably need a clear definition of the most important quality criteria for this product, and then you need a theory that relates test results to the definition. For example, *reliability* is not just about the number of bugs in the product. It is (or is often defined as being) about the number of reliability-related failures that can be expected in a period of time or a period of use. (*Reliability-related?* In measuring reliability, an organization might not care, for example, about misspellings in error messages.) To make this prediction, you need a mathematically and empirically sound model that links test results to reliability. Testing involves gathering the data needed by the model. This might involve extensive work in areas of the product believed to be stable as well as some work in weaker areas. Imagine a reliability model based on counting bugs found (perhaps weighted by some type of severity) per N lines of code or per K hours of testing. Finding the bugs is important. Eliminating duplicates is important. Troubleshooting to make the bug report easier to understand and more likely to fix is (*in the context of assessment)* out of scope.
- ***Verify correctness of the product.*** It is impossible to do this by testing. You can prove that the product is *not* correct or you can demonstrate that you didn't find any errors in a given period of time using a given testing strategy. However, you can't test exhaustively, and the product might fail under conditions that you did not test. The best you can do (if you have a solid, credible model) is assessment--test-based estimation of the probability of errors. (See the discussion of reliability, above).

The primary reason to test is to prove the program works correctly?

Glen Myers vigorously attacked this view in *The Art of Software Testing* and so did I in *Testing Computer Software*.

The problem from one viewpoint is that programmers are offended by testers who express a "negative mindset." I've heard this argument most recently from several programmers in the XP community. Before then, for 15 years, I heard it from more traditional programmers and project managers. A negative attitude causes poor morale. It causes testers to look at the wrong things.

The problem from the other viewpoint is that people are more likely to find what they expect to find and to miss what they aren't looking for (Rosenthal, 1966; Teasley, Leventhal, Mynatt, & Rohlman, 1994). The "positive viewpoint" (prove it works correctly) is an ineffective viewpoint for testers. I've pointed this out to programmers who object to testers' negative mindset and their typical answer is that they don't believe it, they don't care, it does't matter, and it's the wrong attitude for unit testing.

About the unit testing, I think they're right. I teach a course in unit testing and focus my students on building tests to check that the code does what they intend and that serve later as change detectors.

But as to system testing, I think the programmers who dismiss the psychological research without examining it are stubborn ignoramuses.

There is a different contex for the view that the purpose of testing is proving that the program is correct, i.e. proving that the program conforms to the specification. In contract-based development, one can rationalize that any misbehavior not prohibited by a specification is acceptable, and testing for such misbehavior is out of scope of the contracted work. *This might sound irresponsible or unreasonable, but is it unreasonable if* ***the client*** *knowingly agreed to this scope for your work and pays for your labor on that understanding*?

Testers are THE advocates of quality on a project?

I used to hear this as accepted wisdom. It still crops up, even in the most surprising places.[1]

When testers run around telling everyone that "*Someone has to care about the customer*" and "*We are the only ones who care about quality,*" it excludes everyone else. The tester is saying that no one else but testers care about quality. This is incorrect, insulting, demoralizing to nontesters who have been quality advocates, and an excuse for anyone else on the project who wants to say, "*Let those testers worry about quality—we have work to do.*") The worst consequence of this irresponsible rhetoric is that it can become a self-fulfilling property.

Test groups should evolve into quality assurance groups?

I'm not sure what a true quality assurance group is.

- I often think it's other name is "management" because management has the ability to inspire staff to create high quality products and provide high quality services and the power to facilitate, fund, and demand quality.
- Other times, I think the QA group's name is "staff" because I've seen remarkable products come from companies with indifferent executives. I look back to the go-go days of Sili Valley, when companies (and their big investors) would dump empty suits into high executive slots—technocretins who couldn't type their own name at a keyboard, let alone appreciate good work if anyone ever showed it to them, but who knew how to manipulate apparent sales, apparent revenues, apparent debts, in ways that made the balance sheets look good for another quarter—the high quality of products from many of these companies came from staff who respected themselves and their craft, not from their rotating-door management.

Whatever QA is, it's not testing.

Testing is a technical investigation of a product, done to expose quality-related information. Testers dig up useful information. This is good. But it doesn't ensure quality.

[1] A previous draft of this paper inaccurately referenced Crispin & House (2003) at this point. My impression was based on an early draft of that book and that impression is incorrect.

Some people add other types of data collection and analysis to the test group's role and call that QA. It's not. Assuming that the metrics published are valid and useful, this might be a good class of work to add to a test group—more investigation, more reporting. But finding facts doesn't mean acting on facts. You need both to ensure quality—facts and responsive action. Plus proactive action that prevents bad facts from happening in the first place.

Other people add "process management" and say that real QA groups define development (or corporate) processes and ensure that they are followed. This might be closer to the mark, but in my experience, the processes that mattered were either defined by executives or were based closely on policies laid out by executives—and enforced by executives.

Johanna Rothman put this really well in a talk that I'll have to find again (so I can properly cite it). She said that when testers tell her that they do "quality assurance", she asks questions like these:

- Do testers have the authority and cash to provide training for programmers who need it?
- Do testers have the authority to settle customer complaints? Or to drive the handling of customer complaints?
- Do testers have the ability and authority to fix bugs?
- Do testers have the ability and authority to either write or rewrite the user manuals?
- Do testers have the ability to study customer needs and design the product accordingly?

If not, the quality of the product and of the complaining customer's experience in dealing with it, are not under the testers' control.

When I was hired at WordStar, back in 1983 (when WordStar was the top producer of word processing software in the world), the test group was called Quality Assurance. The company was attached to the label *QA* for testers, but as Testing Technology Team Leader, I was able to convince them to change the name, from *Quality Assurance* to *Quality Assistance*.

Quality Assistance--that's what testers do. We help. We investigate. We learn things. We report them clearly. We make sure that people understand what we have found and what its significance means. We provide the good data that is so important for understanding and improving the quality of the product. That's important, but it's not "quality assurance."

The rest of the organization has to do its share or there won't be high quality. We should be inviting other people to show their enthusiasm for improving the product's quality, encouraging them to be aware of their role in quality assurance, not claiming that turf as our own.

The test group should have the power to block release if product quality is too low?

This is another bad idea that used to advocated more widely than it is today—though a lot of people still believe in it.

Here are a few of the serious problems with this (apparent) power:

- It takes accountability away from the project manager who is making the development tradeoffs. The final decision is the testers', not the project manager's.

- It lets project managers (and others) play *release chicken*, a game in which everyone knows the product is not ready for release, but each manager in a meeting says "Sure, we can release it" until finally, one of them chickens out and insists that it is not ready. When we vest the final decision in the test group, we insist that the test group be the chickens in the release chicken game. This sets up the test group to be the bad guys who always hold the release (and blowing the schedule)
- It sets testers up to be “the enemy” without any long-term quality benefit, and it makes testers the ones to blame if the product fails in the field.
- The test group may well lack the knowledge to make the right decision

Another critical fact to recognize is that even testers who have been granted a veto don't really have the power to block a release. They have the power to delay a release. The terms of their veto are that this product will not ship until

(a) the problem is fixed or

(b) a manager or executive with more authority overrides the tester's block.

So the critical question is this—is an apparent veto the most effective way to drive a serious concern about quality to the active attention of a senior manager?

I would rather leave the release decision with the project manager (which is where it belongs) and preserve my authority to send reports and evaluations to the project manager, her boss, and whoever else within the company who I consider appropriate. If the key stakeholders are advised of a problem and accept responsibility for their decisions, they will either make defensible decisions or their replacements will.

Testers and programmers have conflicting interests?

This is true and appropriate in the case of an outsourcing model of development, in which the programming is done by a vendor, and the test group is either a group of employees of the customer or an independent company retained by the customer to check the work of the vendor.

But the idea is often applied to in-house development, and it doesn't work nearly so well there.

For example, one of the companies I chose not to work for paid programmers and project managers (but not testers) a timely release bonus. The testers got a bonus based on technical support costs in the field. The company believed this set up a healthy conflict and a fair reward structure. After all, programmers should want to ship the product today, while testers should want to ship it "right." You can imagine the power plays this generated.

I've had to work through this conflict plenty of times. I've been in plenty of gripe sessions at conferences, where testers talked about the programmers' drive to release products on time, whether they're good or not. I've met programmers and project managers who have that drive.

Sometimes, over the short term, testers have to play this game.

Over the longer term, the better approach is often to refuse to play. A test group can stick to, and publicize, its role as the technical information provider. Final decisions about quality get made by the project manager. If the project manager operates irresponsibly, the project manager faces consequences.

In my experience, many project managers (or their managers) take ownership of the product's problems and operate responsibly. At this point, the tester is a service provider to them, someone who gets them the information they need to make good decisions. This is a healthier relationship, much more rewarding for everyone, and a lot more fun.

The test group should work independently of the programmers?

In the outsourcing situation, there probably has to be an independent testing effort.

But when applied to in-house development, this often reflects fear, frustration, and a lack of vision:

- fear of being biased away from good tests by the programmers;
- fear of being distracted by programmers' agendas;
- fear of having programmers' work delegated to them;
- frustration at dealing with adversarial programmers; and
- lack of understanding the benefits of collaboration between programmers and testers.

Think of black box testing as being either *functional* or *parafunctional*.

- *Functional testing* focuses on capability of the product and benefits to the end user. The relationships that are especially valuable to foster are between testers and customers (or customer advocates), so that testers can develop deeper understanding of the ways to prove that this product will not satisfy the customer's needs.
- *Parafunctional testing* focuses on those aspects of the product that aren't tied to specific functions. These include security, usability, accessibility, supportability, localizability, interoperability, installability, performance, scalability, and so on. Unlike functional aspects of the product, which users and customers understand and relate to pretty well, users and customers are not experts in the parafunctional aspects of the product. Effective testing of these attributes will often require the testers to collaborate with other technical staff (such as programmers).

But the big news of this decade is not parafunctional testing, it's test-driven development (TDD) (Astels, 2003; Beck, 2003). In TDD,

- the programmer (or programming pair) create a test (run it, show that it fails in the absence of new code), then
- add the simplest code that can pass the test, then
- rerun the test and fix the code until the program passes, then
- refactor the code (revise it to make it clearer, faster, more maintainable, etc.)
- then create and check the next test, add code again, retest with all of the existing tests, fix until all tests pass, refactor, and continue until the code is written.

TDD provides a lot of benefits to the project:

- It provides a structure for working from examples, rather than from an abstraction. This supports a common learning and thinking style, instead of fighting against it (as specification-driven design and programming does).

- It provides examples—unit tests—that show how the program works in detail. This is an important, concrete communication with future maintainers of the code.
- It provides a unit-level regression test suite. These tests are often not very powerful, but they are useful because you run them every time you change the code. If any tested assumption is invalidated or violated by added code, the tests serve as *change detectors*. An extensive unit test library provides powerful support for refactoring (Fowler, 2000) and for later code maintenance (bug fixes or additions to existing functionality).
- It makes bug finding and fixing much more efficient.
    - When the programmer finds a bug during TDD, she sees the problem immediately, the unit test identifies the place of failure, and the code is fixed right away.
    - Contrast this with a black box test: the tester finds a problem, replicates it, troubleshoots it to simplify it and understand how to characterize it more effectively, writes the bug report, someone reviews the bug report, it goes to a programmer who might or might not be the right one, eventually reaches the programmer who made the change several days or weeks ago and now has to reconstruct what she was doing, the bug gets fixed (or not), it has to be retested and finally closed. This is far more work per bug.

Brian Marick is one of several members of the agile development community who advocates pairing testers and programmers, to help the programmers make their test-driven tests more effective.

Clearly, with all of the benefits that TDD provides, if the programmers are willing to do it, the testers should be willing to help them.

Testers should push their project teams to follow "appropriately professional" development models, like the Waterfall, that require people to THINK before they act?

Under the Waterfall life cycle, the development team starts by researching and writing the product requirements, then (when these are done) writes the high-level design documents, then (when high-level design is complete) writes low-level design documents, then writes code, then runs the system tests. In a variation of the waterfall, called the V-model, the test group writes its acceptance tests in response to (or in parallel with) the requirements, writes its system tests in response the high-level design, and the testers or programmers write integration tests in response to detailed design, and unit tests in response to (or parallel with) the coding (Goldsmith & Graham, 2002).

These models are often harshly criticized as being inherently high risk (e.g., Krutchen, 2000). The problem is that they force the project team to lock down details long before the implications, costs, complexity, and implementation difficulty of those details are known. The iterative approaches (spiral, RUP, evolutionary development, XP, etc.) are reactions to the risks associated with premature misspecification.

Remarkably many testers still push the traditional waterfall. I don't understand why.

One factor is that waterfall development appears to make it easier to implement GUI-level automated tests because, in theory, the user interface is locked down relatively early. The fragile nature of much of the GUI-level automated test code results in high costs if the program under test has significant user interface changes. However, user interfaces often change very late. If you send code out for beta testing, for example, the user-testers will often ask for plenty of changes to the user interface. Many of those will be well-justified, will be made, and the GUI tests will have to be redone.

A different way to think about the life cycles is to consider the tradeoffs that project managers have to make. The project manager has to deliver a set of features, that work reasonably well, on time and within budget.

Under the waterfall, the features are designed up front, funding and time are spent on designing and developing all of the features, until finally, the code comes into system testing. If the code is late, or the product has a lot of bugs, we face the typical tradeoff between reliability and time to market. (The features are already coded, so even if we cut some, we don't save much time or money.) The testers want more time to test the product and get it fixed, while other stakeholders are likely to argue that it is time to ship the product before it runs too far behind schedule.

Iterative life cycles are designed to change this tradeoff. For example, in an evolutionary model (eXtreme Programming follows an evolutionary model, but evolution was in practice long before XP (Gilb, 1988)), you test each feature as you add it to the product, and bring it up to acceptable quality before adding the next feature.

In the evolutionary case, the tradeoff is between features and time to market, rather than between reliability and time to market. Testers play a much less central role in the late-project controversies because the issue is not whether the product works (it does) but instead is whether there's enough product.

Testers should base test cases on documented characteristics of the program. If the software documentation is inadequate for this, testers should assert a quality control function and demand better specification and requirements documentation?

My experiences and disagreement with these two assertions played a key role in motivating me to start writing *Testing Computer Software* in 1983. Twenty years later, these assertions are still in play. Not so long ago, I saw a leading consultant/author publicly tell a pair of newly-promoted test managers to refuse to test software if it came without adequate specifications. The test managers asked whether they could really refuse, and the consultant explained that this was *their professional duty*, because it is unprofessional to develop software without detailed specifications.

My first problem with guidance like this is that a tester whose test design and test planning process depends on a strong specification will be helpless in the face of a lightly specified product.

- Many development groups choose not to write detailed specifications. *Is this always bad?*
- Many product changes are made without updating the specification, because the spec is not considered final authority in that company. *Is this always bad?*

Testers should base their tests on the information they can get—and they can get *plenty* of information from sources other than specifications.

My second objection is that specifications describe how the program is *supposed to* work. Testers discover how the program *doesn't* work. The specification isn't necessarily the best source of information on product risks.

My third objection is that this approach to testing, in practice, narrows the range of the tester's thinking. Testers often write only one test per specification item—because what we focus on is covering every statement in the specification, rather than questioning all of the different ways in which the code associated with any given statement could go wrong.

A specification is only one source of guidance, fallible guidance, about what to test and how the program should work. To the extent that a specification limits what you test, it is a source of risk.

My final objection to this guidance is that a tester who refuses to proceed until the engineering process is changed is hijacking the management of the project. If you want to manage the project, become a project manager. But if you're a service provider to a project, you are not the project manager. Refusal to deliver critical services should be done only as a last resort under extreme circumstances. People who make a habit of it, in my experience, get fired. As, in my opinion, they should.

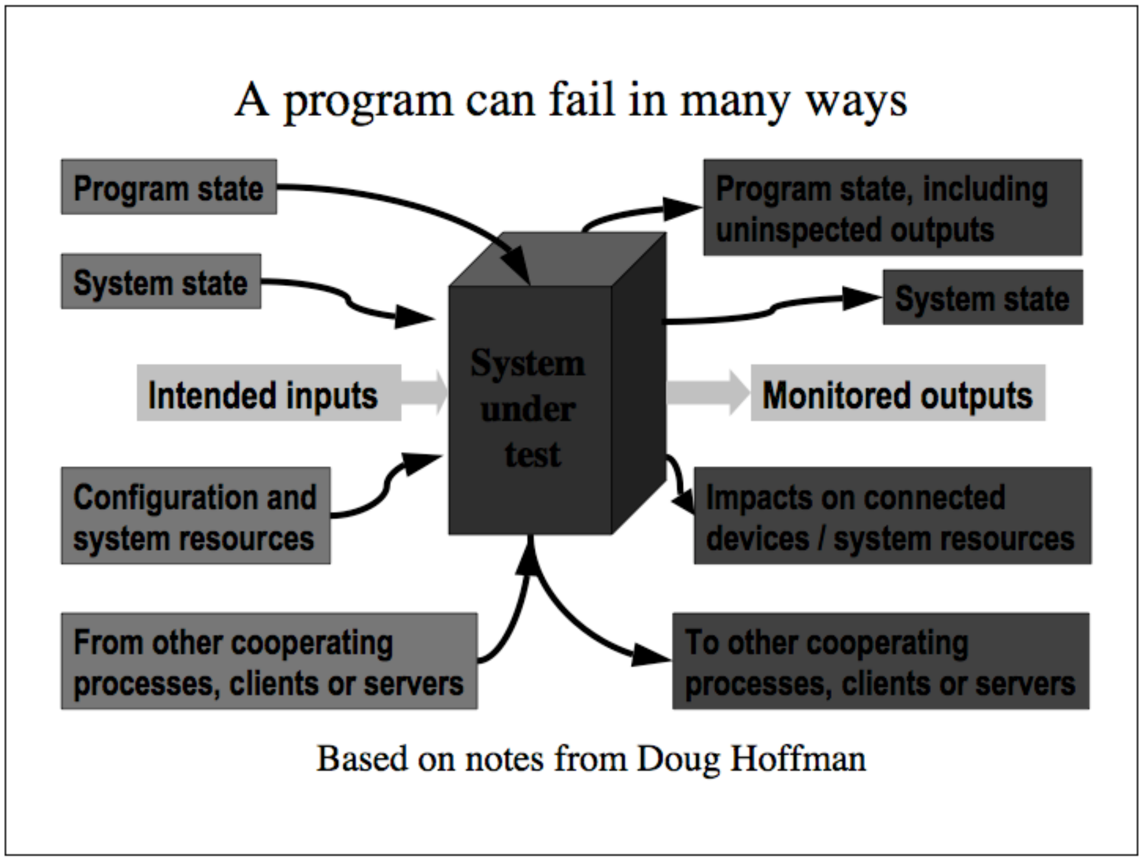
Test Planning and Documentation

Testers should specify the expected result of every test, in advance?

This is another guidance from Myers (1979) that has had lasting influence. See, for example, ISEB's current syllabus for test practitioner certification at www1.bcs.org.uk/DocsRepository/00900/913/docs/practsyll.pdf.

One fundamental problem with this idea is that it is misguided. Every test has many results. No one specifies them all. An "expected result" points the tester at one or a few of these results, but away from the others.

For example, suppose we test a program that adds numbers. Give it 2+3 (the intended inputs) and get back 5 (the monitored output). This looks like a passing test, but suppose that the test took 6 hours, or that there was a memory leak, or that it erased a file on the hard disk. If all we compare against is the expected result (5), we won't notice these other bugs.


A program can fail in many ways
Program state
System state
Intended inputs
Configuration and system resources
From other cooperating processes, clients or servers
System under test
Program state, including uninspected outputs
System state
Monitored outputs
Impacts on connected devices / system resources
To other cooperating processes, clients or servers
Based on notes from Doug Hoffman

The problem of an observer (such as a tester) missing obvious failures is not just theoretical. I've seen it at work, and psychologists have described it in research, naming it *inattentional blindness* (Mack & Rock, 2000). If you are focusing your attention on something else, then you have a significant probability of completely missing a surprising event that happens right in front of you. Some striking demos are available at http://viscog.beckman.uiuc.edu/djs_lab/demos.html.

Does this mean that it is a *bad idea* to develop expected results? No, of course not. It means that there are pluses and minuses in using, and relying on, expected results.

Expected results are valuable when an answer is difficult or time-consuming to compute. They are valuable when testing is done by junior staff who don't know the product (*but realize that these people are especially likely to miss other failures)*. They are valuable for the test planner who is thinking through how the product works while she design the tests—working the tests through to their predicted conclusions is an important tool for learning the product. They are of course valuable for automated tests—but the ability to automatically *derive* an expected result is much more valuable for automated testing (because it is *so much faster*) than the ability of a human to write the result down.

My second issue with the requirement that all tests should be specified up to and including their expected results is that it bars many of the important types of high volume automated testing (Kaner, 2000; Kaner, Bond, & McGee, 2004; Nyman, 2000). These tests expose memory leaks, wild pointers, stack corruption, timing errors and many other problems that are not anticipated in the specification, and are clearly inappropriate (*i.e. bugs)*. Running huge searches that are in some ways random exposes interesting problems that we often don't know how to effectively search for more directly. The only "expected results" might be not crashing, or nothing-bad-happens-here. Because they are so general, these results have little useful relationship to the actual risks we attempt to mitigate with sequences of tests like this

My final issue with the *requirement* that all tests should be specified up to and including their expected results is that it pushes the notion of *testing as confirmation* over the vision of *testing as investigation*.

- The confirmatory tester knows what the "good" result is and is trying to find proof that the product conforms to that result.
- The investigator wants to see what will happen and is expecting to learn something new from the test. The investigator doesn't necessarily know how a test will come out, how a line of tests will come out or even whether the line is worth spending much time on.

It's a different mindset. Some people describe the investigative mindset as "experimentation" rather than "quality control." OK. The suggestion they're making is that we should be quality controllers rather than experimenters. Good for them. But I think we should be experimenters, and that's the best way we can learn the most that we can learn, to help assess and improve product quality.

Exploratory testing, if done at all, should be done only by experts?

Exploratory testers learn about the product and its risks while they design and test the product. Testing, learning and design are parallel, interacting processes (Bach, 2002; Kaner & Bach, 2004a).

Exploration is the opposite of following a preset plan with specified tests and specified test results. It is viewed with suspicion in traditional quarters, and even some untraditional ones (Crispin & House, 2003). For example, the IEEE SWEBOK (Software Engineering Body of Knowledge) (Committee, 2004) recommends that it be done only by experts.

James Bach and I have been working with exploratory testing throughout our careers. I coined the phrase in 1984, published it in (Kaner, 1988) with the strong recommendation that testers should run exploratory tests in every cycle of testing.

Since then, I've come to realize that, in practice, good testers explore all the time. For example, when you report a bug and are told that it's been fixed, you do regression testing. Junior testers run exactly the same steps they used the first time and accept the bug as fixed if it passes the exact original test. Testers with more experience retest with their original steps but then vary the steps. They test around the bug, checking the generality of the fix and looking for side effects. How long they test, what ways they vary their tests, what risks they entertain while they test, are decisions that testers make as they go, as they see what results they've gotten with the tests they've run so far.

Being willing to change what you do, to follow up what you're seeing as you test, is the mark of a thoughtful tester. A tester who won't create and run follow-up tests in the face of an oddity in the program behavior is a tester who needs training, retraining, or a career change—after all, if you go to all the time and effort to run a long series of tests, and one of them shows something that might be interesting, you're throwing away the value of all that work if you don't check the interesting behavior to see whether it's a bug and how best to report it.

Scenario testing is another style of testing that is often done in an exploratory way. A scenario test is test couched in a story about the use of the product that reflects real-life use and that would cause an influential stakeholder to protest if the program fails the test. Scenario tests typically check whether a given benefit is actually provided by the program, or is too hard or too confusing to reach. You might spend a long time doing the research behind a scenario test or series of them (Kaner, 2003a), but many people will only run a scenario test once or twice because it usually provides most of its value (insight into the program's design and the users' requirements) the first time it's run. Developing scenarios guides you in your learning about customers, users, and requirements for the product. As you research more, and test your learning against the product, you gain more insight and design more interesting tests.

You can do any type of testing in a preplanned, scripted, routine way or an exploratory way. Some techniques lend themselves more readily to instant variation, but all techniques' tests can be adapted by a tester who is learning as she tests.

There should be at least one thoroughly documented test for every requirement item or specification item?

We looked at this question above, when considering whether it was appropriately within the tester's role to demand a specification (and refuse to test if no spec was forthcoming).

Suppose you have a specification. What should you do with it?

A specification is only one source of guidance, fallible guidance, about what to test and how the program should work. There are many, many implicit specifications that describe user expectations even though they are not part of the official spec. For example:

- The rules of mathematics normally describe what users expect (or should expect) of the mathematical operations of a program, and that's true whether someone says "mathematics works as usual" in the specification or not.
- The user interface conventions of a platform are often authoritatively described by the platform vendor or an influential third party.
- The ways in which the product would normally be used might be most thoroughly described in books or professional courses—for example, accountants learn a lot more about accounting in their accounting courses than from the specification or user manual of a program they're using. And if the program fails to conform to their education-based expectations, they're probably going to mistrust the program more than their course or published specs.

In my experience, this approach to testing, narrows the range of the tester's thinking. Testers ignore the implicit specs in order to do a good job on the explicit ones. And testers often write *only* one test per specification item—because what we focus on is covering every statement in the specification, rather than questioning all of the different ways in which the code associated with any given statement could go wrong.

Testers should design most or all tests early in development?

Advocates of the waterfall (and its most current variant, the V-model) would have us design most tests early in development.

Traditional Development

I think this is mistaken, in traditional development, for several reasons, including these:

- One of the core problems of testing is the infinity of possible tests. Good testing requires selection of a tiny subset from this pool of tests. The better we understand the product and its risks, the more wisely we can pick those few tests. At the start of the project, testers are at their most ignorant. This is when they *start learning* about the product and its risks and they will *keep learning* as the project continues.
- As the program is developed, programmers make mistakes. Different programmers make different kinds of mistakes. As testers work with the product, they gradually learn which types of mistakes are more common, in this product, from these programmers. You can't know that before the code is written. You can't know that at the start of the project. You learn from experience. If you design your tests without this knowledge, you are blind to the risks that you will come to understand later.
- When you invest in early development of tests, you've spent time and money on all these tests. If the program changes, you have to redo your work. So what happens when the product goes out for beta testing and the users say they hate parts of it? Are you going to be part of the lobby pushing to fix it, or part of the lobby saying it will cost too much to make this quality improvement, let's just ship it.

Project Inertia

Agile developers often write about a program's *velocity*, the rate at which they can add working functionality (Jeffries, Anderson, & Hendrickson, 2001). The counterpoint to velocity is *project inertia*, the resistance to change that we build into a project.

Some inertia is built into a project intentionally. For example,

- Some projects use change control boards to review every change (Pressman, 2005).
- Many projects impose a user interface freeze, locking down the visible behavior of the project.

Other inertia arises out of costs of change, such:

- The cost of changing the specification
- The cost of rewriting code
- The cost of rewriting the user documentation
- The cost of testing changes to code for effectiveness and for side effects
- The cost of revising the test plan and testing materials
- The cost of rerunning all the regression tests

The earlier we make key decisions or create key artifacts, the greater the cost of later change, and so the higher the inertia.

Does this mean we should not design or document tests? No. It means that we should do the work as and when it is needed, locking down details as late as possible.

Agile Development

The context is quite different for most agile projects, because code is designed, created, and tested in rapid iterations. Features that are created early are tested early, as they should be.

Testers should design all tests for reuse as regression tests?

As above, it is imperative to distinguish between unit tests and system tests. Unit tests are almost always automated, and it makes a great deal of sense to create a large pool of change detectors. I think the recent advances in unit testing (Astels, 2003; Rainsberger, 2004) have been the most exciting progress that I've seen in testing in the last 10 years.

With programmer-created and programmer-maintained change detectors:

- There is a near-zero feedback delay between the discovery of a problem and awareness of the programmer who created the problem
- There is a near-zero communication cost between the discover of the problem and the programmer who will either fix the bug or fix the test
- The test is tightly tied to the underlying coding problem, making troubleshooting much cheaper and easier than system-level testing.

But most of the advice to make all tests regression tests, preferably automated regression tests, has referred to user-interface level system tests. This is another old, bad idea that I've been surprised to see resurrected in some parts of the agile community (Crispin, 2001).

We have a limited amount of time and an unlimited amount of testing work to do. So, we must select tests intelligently.

Automated UI level regression tests are not free. They take time to create and a lot more time to maintain as the program under test changes (Kaner, 1998; Marick, 1998a).
With automation, or non-automated reuse of exising tests, we have a cost/benefit analysis to consider carefully:

- What information will we obtain from re-use of this test?
- What is the value of that information?
- How much does it cost to automate the test the first time?
- How much maintenance cost for the test over a period of time?
- How much inertia does the maintenance create for the project?
- How much support for rapid does the test suite provide for the project?

In general, tests that have not exposed bugs previously are not strong candidates for exposing bugs in the near future.

There are good reasons to create *some* automated system-level regression tests, such as (among others):

- smoke tests (tests that expose obvious problems, run to qualify a build for more in-depth testing)
- tests required or desired for regulatory inspectors, and
- tests of aspects of the product that seem particularly vulnerable to side effects.

But that suggests that we should adopt a balanced strategy, with some regression tests and some (or many) single-use tests.

Testers should document manual tests in great procedural detail so that they can be handed down to less experienced or less skilled testers?

The claim is that manual tests written as detailed scripts can be handed to less experienced or less skilled testers, who will:

- repeat the tests consistently, in the way they were intended,
- learn about test design from executing these tests, and
- learn the program from testing it, using these tests.

The idea has been popular, especially with people who promote outsourced testing (to local or faraway test labs). It is so popular that James Bach, Bret Pettichord and I identified it as a basis of one of the four dominant schools of software testing, the "Factory School" (which Pettichord (2003) now names the Routine School).

However, there are some problems:

- It takes a long time to document and maintain the written procedures. The problems that I raised with automated GUI-level regression testing apply even more to manual regression testing because the manual work is, in my experience, less efficient. All the same cost-benefit tradeoffs apply (but less favorably) as do the problems with project inertia (worse, because written test scripts are harder to maintain than test scripts in code—the code can tell you its wrong when you execute it, whereas you have to hunt for the errors in the written scripts).
- In my experience supervising them, junior testers are more likely to be influenced by the specified expected results. As they follow the test script like little robots, they are more likely to miss suspicious behaviors and even obvious failures that were not called out for attention in the test script.
- Finally, there is no reason to expect these scripts to be effective training / learning tools, no evidence published that they have ever been any good as training tools, and plenty of evidence from the study of science education that indicates that scripted practice would be a pretty ineffective way to teach math or science (or testing) (Bransford, Brown, & Cocking, 2000; Lesh & Doerr, 2003). Despite my initial enthusiasm for the approach, it has never been a good way to teach me or my staff anything useful about test design or the program.

I liked this idea when I was going to university. It made sense to me because I was used to packaging tasks for others to do in other types of work that I had managed. It took about a year of supervising testers before I began to get seriously suspicious of the idea. Since about 1994, I've been describing it as an *industry worst practice*.

Testers should document each test case individually, ideally storing them in a test case management system that describes the preconditions, procedural details, postconditions, and basis (such as trace to requirements) of each individual test?

The piece that I'm highlighting here is the test case management system. I keep seeing systems that store each test case separately, with a separate script for each test, a separate description of preconditions and expected results for each test, etc.

We often design tests in themes, a group of tests that are governed by the same basic idea but are variations of each other. One approach to documenting these is to describe the base idea and the general ideas for variation. For example, look at the test matrices in my course notes (Kaner & Bach, 2004b). A much more time-consuming approach, with much higher maintenance costs, is to separately specify each test in its own lavish detail.

The Practice of Testing

Testers should assume that the programmers did a light job of testing and so should extensively cover the basics (such as boundary cases for every field)?

This is a standing assumption for many software test groups. I've worked with programmers for whom this was appropriate, and I've worked with programmers who delivered much more thoroughly tested code than this.

One critical problem with this approach is that testers spend so much time on the simple qualification tests that they run out of time before they can get into the more complex tests that look at how experienced users will work with the product or how the product handles interesting data sets. A test group that is stuck with lightly tested code might consider specializing one or a few testers on scenario testing and other more complex tests, so that this work is done in parallel with the basic testing. The product probably won't be tested well enough in the basic or the advanced tests this way, but you'll find significant problems (if they're there) and have (and be able to report) a clearer understanding of the quality of the code.

A second problem shows up when testers insist on this approach in the face of evidence that the programmers are doing a reasonable job of unit testing. In this case, many of the system testers' tests have already been much more efficiently done by the programmers.

Testers should develop automated tests at the user interface level?

Why automate GUI-level tests?

Rather than starting from the assumption that most of our GUI-level tests *should* be automated, perhaps we should be asking how to *avoid* automating them?

GUI tests are problematic because they create user interface inertia—they are tied to details of the user interface, and so either the UI has to stop changing or the testers have to revise the tests over and over again.

Three alternatives for test automation are at the unit level, an integration level that is still tied tightly to the code (Ward Cunningham's FIT framework, at http://c2.fit.edu), and application programmer interface (API) test level.

It may well be that even if you push many of your automated tests down to unit, integration framework, or API levels, there will still be significant need for GUI-level automated tests. But at least in this case, the only ones you'll create at that level (and at that price) will be the ones you couldn't reasonably do in any other way.

Testers should design tests without knowledge of the underlying code?

Why should testers design tests without knowledge of the code?

One common answer is that this ignorance protects testers from bias? Huh? What bias? What protection?

A different possibility is that if testers don't need to know the code, maybe it's OK to hire testers who can't code.

I think a better reason is discipline—rather than learning about the product from the code, the black box tester has to learn from other sources of information, such as sources that describe how the product is used, how products like this fail in the field, etc. Often, these are sources that the programmer didn’t consider.

The value of a black box approach is not freedom from programmer influence. It is a refocusing of the tester on the needs of the user and the risks in the market.

The critical thing to recognize is that the decision is a practical one, not a matter of principle:

Contrast this:

> *What does the project gain from a tester who focuses on customer benefits, configuration / compatibility, and coordination with other products?*

with this:

> *What does the project gain from a tester who can review code to determine plausibility of some tests, and can implement code-aware test tools / techniques (e.g. FIT, simulators using probes, event logs, etc.)*
>
> *What does the project gain from a tester who actively collaborates with the programmers in the analysis and debugging of the code?*

The best choice will depend on the circumstances of the project and the skills and interests of the staff.

Most testers don't need to be programmers, because they only have to specify how the external program will behave in their tests. Testing

tools will allow business experts to develop automated tests without having to program?

We've been hearing for years that the latest test tools have super-friendly features (like capture/replay) that allow nonprogrammers to design and implement great tests.

Yet, at conferences, I hear the reverse. Testers stand up and tell us in meeting after meeting that the simple scripting solutions failed, and more complex solutions involving homebrew or home-modified test automation frameworks made maintainable test case implementation possible. This is not work for non-programming business experts who lack sophistication in testing.

Business experts have a great deal to offer the test planning and test execution process. They can offer realistic, challenging tests, but they're generally focused on the idea of a product that (they hope) works. Their tests are more likely to be confirmatory or to focus on specific failures they've had to recover from before. Testers—what I think of as *good testers*—have a broader set of ideas about how products fail and can think along several dimensions—not just functionality, but also security, usability, performance, and the several other parafunctional dimensions of product quality.

Testers should design the build verification tests, even the ones to be run by programmers?

This is an idea from the 1990's rather than the 1960's. As testers begain automating their build verification tests, they started suggesting to programmers that they (programmers) run the test suites to qualify builds before delivering them to the testers.

The idea sounds plausible, but ask yourself what build verification the programmers *would have done* if they weren't using suites from the testers? The programmers have a very different understanding of the code and its risks from the black box testers, and programmers find a high percentage of their own bugs. If we substitute tester-thinking for programmer-thinking at build verification, when will we get programmer-designed integration and system integrity tests?

The pool of tests should cover every line and branch in the program, or perhaps every basis path?

I've addressed this assertion in detail elsewhere (Kaner, 1996, 2001; see also Marick, 1997).

Monitoring our coverage of lines of code is simplistic. There are plenty of other things to count besides lines, such as dataflows or anticipated failure modes (I listed 99 examples in 1996). In addition, focusing testing on executing program statements can miss a lot of potential problems—imagine executing a line that contains A/B. The program might pass the test if B is not zero and fail (crash, at least) if B is zero. There are plenty of potential problems that will be missed or exposed only by luck if the tester's primary design criterion is working through every line of code.

As Marick (1997) pointed out, when managers press staff to meet some requirement, such as testing every line, the staff are likely to sacrifice other test attributes in order to meet the stated requirement as directly as they can. This is a problem of performance measurement and management in general (Austin, 1996).

A different class of problem for black box testers is a practical one. Code coverage measurement makes sense in the unit testing context. In the system-level, black-box testing situation, the tester doesn't know the code, probably has no access to the code, and probably lacks the tools, knowledge and skill to identify and reach those last few untested percent of lines of code.

All failures must be reported into a bug tracking system and carefully counted?

What does this mean?

Does "all" failures mean *all?* Do programmers have to enter every bug they make (and then discover and correct) into the tracking system? Even spelling mistakes and other typos? That's not often done.

If a tester is loaned to the programming team to help them do their testing, and they're at a point where they wouldn't normally start entering bugs into the tracking system, should the tester break their convention and report what she finds anyway? Some of us think that's inappropriate (Kaner et al., 1993; Marick, 1998b).

Many testers would agree that the early bugs don't need to be reported, but once regular testing has started, there are strong feelings that bugs found by testers must go into a bug tracking system. At some testing conferences, the Extreme Programmers' practice of writing bug reports on 5x7 cards has been the subject of particularly strong criticism—by people who had never tried XP.

The core purpose of a bug tracking system is to get the right bug fixed.

If an informal, manual system achieves this, might we be better off without the more formal system?

The difference in context is that in XP, bugs are fixed as they are found. The stack of open bugs is never very large. And part of every bug fix is creation of regression tests that check the fix. In that context, perhaps some of the key reasons for a more formal system have been controlled in other ways. If so, then failure to use a database would not reflect unprofessionalism, a sloppy mentality, or a desire to evade accountability. It would reflect a preference for efficiency over unnecessary process investments.

We can measure the individual effectiveness of software testers by counting how many bugs they find, perhaps with an adjustment for bug severity?

Two testers test for a month each. One (call him T100) reports 100 bugs, the other (call her T10) reports 10. The severities are comparable. Which one is the better tester? Before you say it's obvious that T100 is better, what if T10 tested code that was much more stable and found bugs that were much harder to find? What if T10 tested code that implemented quantum physics computations and T10 had to do significant research to figure out what was worth testing, how and why? What if T10 tested code, or tested against risks, that required more extensive test setups?

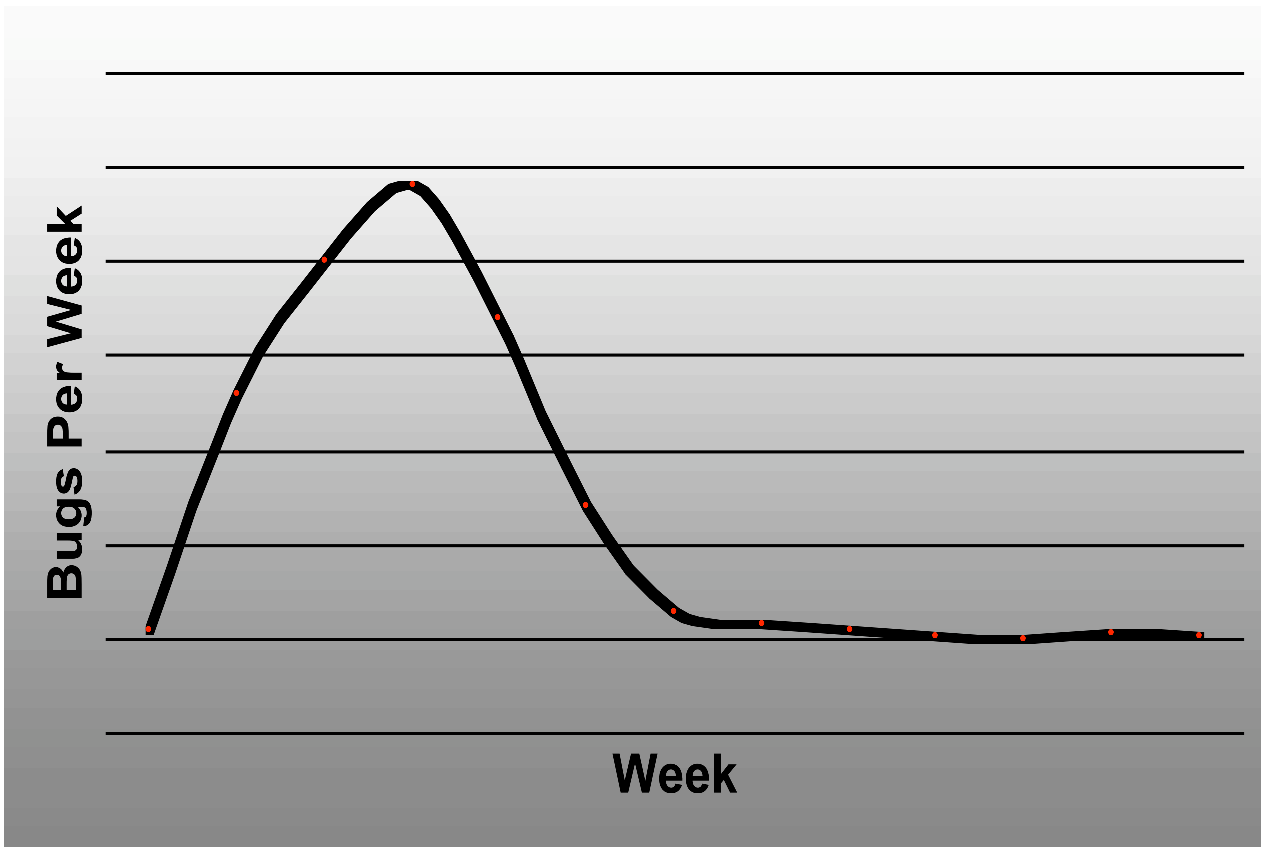
Is bug finding the only thing that testers do? If we reward testers primarily for bug reports, what effect would that have on their willingness to develop test tools, help tech support staff understand unfixed bugs and their workarounds, or do other tasks that don't lead directly to higher bug counts?

Bugs are easy to count but they tell us little about the quality of the tester's work, and counting them distorts that work in ways that are likely to be unhealthy (Kaner, 1999, 2001).

We can tell how close we are to release of the product by examining the bug curve that shows bug finds per week?

One of the key release criteria for a project is an acceptably low count of significant, unfixed bugs. It is common, over the course of the project, for testers to find a few bugs at the start (while they're getting oriented), then lots of bugs, then fewer and fewer as the program stabilizes. The pattern is common enough that bug curves—graphs showing how many new bugs were found week by week, or how many bugs are unresolved week by week, or some other weekly variant—are in common use in the field.

Some companies go further, fit a probability model (typically a reliability model) to the curve, and try to estimate when (in the future) they will be able release a product by fitting the bug counts to the curve.



A typical example uses the Weibull model, which involves the following assumptions (Simmons, 2000):

1. The rate of defect detection is proportional to the current defect content of the software.
2. The rate of defect detection remains constant over the intervals between defect arrivals.
3. Defects are corrected instantaneously, without introducing additional defects.
4. Testing occurs in a way that is similar to the way the software will be operated.
5. All defects are equally likely to be encountered.
6. All defects are independent.
7. There is a fixed, finite number of defects in the software at the start of testing.
8. The time to arrival of a defect follows a Weibull distribution.
9. The number of defects detected in a testing interval is independent of the number detected in other testing intervals for any finite collection of intervals.

Simmons points out that some of these assumptions are unrealistic, but dismisses the problem:

> These assumptions are often violated in the realm of software testing. Despite such violations, the robustness of the Weibull distribution allows good results to be obtained under most circumstances." (Simmons, 2000, p. 4)

Pat Bond and I (Kaner & Bond, 2004) described the problems with these assumptions in more detail.

Doug Hoffman (2000) pointed out dysfunctional consequences of using models like these in companies that he had consulted to. I've seen many of the same problems.

When a company operates on the assumption that this model actually predicts its release dates, managers work to try to make the results look a little better. They often create some rewards and punishments to encourage testers and programmers to do the "right" kinds of work at the right times.

For example, early in testing, the project team might be pressured to increase their bug counts. After all, the faster they rise to the project peak, the more right-skewed the curve and the faster we would predict the curve would drop to low numbers (and an early release). In response to the pressure, testers have been known to do some odd things to bring up the early bug counts, such as (Kaner & Bach, 2004b):

- Run tests of features known to be broken or incomplete.
- Run multiple related tests to find multiple related bugs.
- Look for easy bugs in high quantities rather than hard bugs.
- Put less emphasis on infrastructure, automation architecture, tools and more emphasis of bug finding. (These give them a short term payoff but long term inefficiency.)

Later in testing, the pressure is to decrease the rate of finding new bugs. In conformance, testers or programmers might:

- Run lots of already-run regression tests
- Not look as hard for new bugs.

- Shift their focus to appraisal, writing status and quality reports.
- Classify unrelated bugs as duplicates.
- Class related bugs as duplicates (and closed), hiding key data about the symptoms / causes of the problem.
- Postpone bug reporting until after a measurement checkpoint (milestone). (Bugs have sometimes lost this way because the records kept were too informal.)
- Report bugs informally, keeping them out of the tracking system
- Send the test team to the movies before measurement checkpoints, to slow down their bug find rate (No, I am not kidding about this.)

All this dysfunction comes naturally because the underlying model has so little to do with the reality of the project that making things look better can be done in ways that have nothing to do with making the project better for release.

> A project is complete enough to release when it provides enough of the features, delivers enough of the benefits (the features have to work well enough together for the user to actually succeed in using the product to get real work done), is documented well enough for the user, validated well enough for regulators or other stakeholders (e.g. litigators of the future) who have a legitimate interest in the validation, has been sufficiently instrumented, documented, and troubleshot to be ready for field or phone support, is sufficiently ready for maintenance, localization or porting to the next environment (readiness might include having maintainability features in the code as well as effective version control and other maintainability-enhancing development processes in place), is acceptable to the key stakeholders, and has few enough bugs. This list is not exhaustive, but it illustrates the multidimensionality of the release decision. Many companies appraise status and make release decisions in the context of project team meetings, with representatives of all of the different workgroups involved in the project. They wouldn't need these team meetings if the status and release information were one-dimensional (bug counts). We describe these dimensions in the language of "good enough" because projects differ in their fluidity. One organization might insist on coding everything agreed to in a requirements specification but do little or nothing to enable later modification. Another might emphasize high reliability and be willing to release a product with fewer than the desired number of features so long as the ones that are included all work well. Even if we restrict our focus to bugs, the critical question is not how many bugs are in the product, nor how many bugs can be found in testing but is instead how reliable the product will be in the field (Musa, 1999), for example how many bugs will be encountered in the field, how often, by how many people, and how costly they will be. (Kaner & Bond, 2004)

Conclusions

Many members of the testing community's current vision of testing looks a lot like what I learned in school in the 1970's. For example, take a look at ISEB *Practioner Syllabus* (British Computer Society Information Systems Examinations Board, 2001).

The 1960's and 1970's were a different time. Programming was linear. Programs were shorter, much shorter. Development was often done by outside companies and even if programming was done in-house, testing (if it was done with any formality) was often farmed out. Ideas that worked well for that context don't necessarily stretch well to other contexts.

Professional software development has become much more varied. XP and test-driven development offer a different vision of the relationship between testers and programmers. Even more traditional projects demand better user interfaces—better usability, not just pretty screens—and better coordination among the staff. The larger programs make prioritization and automation even more urgent, and component-based development (which allows programmers to sew together large products much more quickly) puts a disproportionate burden on testing.

With the paradigmatic changes in development, we need shifts in testing as well. This paper doesn't spell out all the important shifts, but it does open many of the established ideas up for reconsideration.

The tone of this paper has been sharply critical of the traditional ideas. Please understand that I'm writing sharply to make a point. Most of these ideas apply perfectly well—in some contexts. What I'm doing here is challenging you to think about the context your in, set aside pre-existing assumptions, and think through what craft and culture of testing will actually work best where you are, or where you want to become.

REFERENCES

Astels, D. (2003). *Test Driven Development: A Practical Guide*. Upper Saddle River, New Jersey: Prentice Hall PTR.

Austin, R. D. (1996). *Measuring and Managing Performance in Organizations*: Dorset House.

Bach, J. (2002). Exploratory Testing Explained. In E. v. Veenendaal (Ed.), *The Testing Practitioner*. The Netherlands: Tutein Nolthenius. http://www.satisfice.com/articles/et-article.pdf

Beck, K. (2003). *Test-Driven Development By Example*. Boston: Addison-Wesley.

Bransford, J., Brown, A. L., & Cocking, R. R. (Eds.). (2000). *How People Learn: Brain, Mind, Experience, and School (Expanded Edition)*: National Academies Press.

British Computer Society Information Systems Examinations Board, I. (2001, September 4, 2001). *Practitioner Syllabus V 1.1*. Retrieved November 21, 2004, from http://www.bcs.org/NR/rdonlyres/C5FCD29A-6F0F-4F56-995D-DDC8846075A2/0/practsyll.pdf

Committee, S. E. C. (2004, June 23). *Guide to the Software Engineering Body of Knowledge (SWEBOK)*. Retrieved November 21, 2004, from http://www.swebok.org

Crispin, L. (2001). eXtreme Rules of the Road: How a tester can steer an eXtreme Programming project toward success. *STQE, 3*(4), 24-30.

Crispin, L., & House, T. (2003). *Testing Extreme Programming*. Boston: Addison-Wesley.

Fowler, M. (2000). *Refactoring: Improving the Design of Existing Code*. Boston: Addison-Wesley.

Gilb, T. (1988). *Principles of Software Engineering Management*. Wokingham, England: Addison-Wesley.

Goldsmith, R., & Graham, D. (2002). The Forgotten Phase. *Software Development*. http://www.sdbestpractices.com/documents/s=8815/sdm0207e/0207e.htm

Hoffman, D. (2000, October, 2000). *The Darker Side of Metrics*. Paper presented at the Pacific Northwest Software Quality Conference, Portland, Oregon. http://www.softwarequalitymethods.com/SQM/Papers/DarkerSideMetricsPaper.pdf

Jeffries, R., Anderson, A., & Hendrickson, C. (2001). *Extreme Programming Installed*. Upper Saddle River, NJ: Addison Wesley.

Kaner, C. (1988). *Testing Computer Software* (1st ed.). New York: McGraw Hill.

Kaner, C. (1996, May 16). *Software Negligence & Testing Coverage*. Paper presented at the Software Testing Analysis & Review Conference (STAR), Orlando, FL. http://www.kaner.com/pdfs/negligence_and_testing_coverage.pdf

Kaner, C. (1998, May 6). *Avoiding Shelfware: A Manager's View of Automated GUI Testing*. Paper presented at the Software Testing Analysis & Review Conference (STAR East), Orlando, FL. http://www.kaner.com/pdfs/shelfwar.pdf

Kaner, C. (1999). Don't Use Bug Counts to Measure Testers. *Software Testing & Quality Engineering, 1*(3), 79-80. http://www.kaner.com/pdfs/bugcount.pdf

Kaner, C. (2000). *Architectures of Test Automation*. Paper presented at the Software Testing, Analysis & Review Conference West, San Jose, CA. http://kaner.com/pdfs/testarch.pdf

Kaner, C. (2001, April). *Measurement Issues and Software Testing (Keynote Address)*. Paper presented at the QUEST 2001 Segue Software User's Conference, Orlando, FL.

Kaner, C. (2003a). The power of 'What If…' and nine ways to fuel your imagination: Cem Kaner on scenario testing. *Software Testing and Quality Engineering Magazine, 5*(5), 16-22. http://www.kaner.com/pdfs/ScenarioSTQE.pdf

Kaner, C. (2003b). *What is a Good Test Case*. Paper presented at the Software Testing Analysis & Review East, Orlando, FL. http://kaner.com/pdfs/GoodTest.pdf

Kaner, C., & Bach, J. (2004a). *The Nature of Exploratory Testing*.Unpublished manuscript, Tampere, Finland. http://www.kaner.com/pdfs/NatureOfExploratoryTest.pdf

Kaner, C., & Bach, J. (2004b, September). *Part 2--Complete Testing is Impossible*. Retrieved November 21, 2004, from http://www.testingeducation.org/k04/documents/bbst9_2004.pdf

Kaner, C., Bach, J., & Pettichord, B. (2001). *Lessons Learned in Software Testing*: Wiley.

Kaner, C., & Bond, W. P. (2004, September 14-16). *Software Engineering Metrics: What Do They Measure and How Do We Know?* Paper presented at the 10th International Software Metrics Symposium (Metrics 2004), Chicago, IL. http://www.kaner.com/pdfs/metrics2004.pdf

Kaner, C., Bond, W. P., & McGee, P. J. (2004, May). *High Volume Test Automation (Keynote Address)*. Paper presented at the International Conference for Software Testing Analysis & Review (STAR East), Orlando, FL. http://www.testingeducation.org/articles/KanerBondMcGeeSTAREAST_HVTA.pdf

Kaner, C., Falk, J., & Nguyen, H. Q. (1993). *Testing Computer Software* (2nd ed.): International Thomson Computer Press.

Kaner, C., Falk, J., & Nguyen, H. Q. (1999). *Testing Computer Software* (2 ed.). New York: John Wiley & Sons.

Krutchen, P. (2000). *The Rational Unified Process: An Introduction*. Boston: Addison-Wesley.

Lesh, R. A., & Doerr, H. M. (2003). *Beyond constructivism: Models and modeling perspectives on mathematics, problem solving, learning and teaching*. Mahwah, NJ: Lawrence Erlbaum.

Mack, A., & Rock, I. (2000). *Inattentional Blindness*. Cambridge, MA: Bradford Books / MIT Press.

Marick, B. (1997). *How to Misuse Code Coverage*.Unpublished manuscript. http://www.testing.com/writings/coverage.pdf

Marick, B. (1998a, May 26-29). *When Should a Test be Automated?* Paper presented at the International Software Quality Week, San Francisco. http://www.testing.com/writings/automate.pdf

Marick, B. (1998b). *Working Effectively with Developers*. Paper presented at the Software Testing Analysis & Review Conference (STAR West), San Diego, CA. http://www.testing.com/writings/effective.pdf

Musa, J. (1999). *Software Reliability Engineering*. New York: McGraw-Hill.

Myers, G. (1979). *The Art of Software Testing*: Wiley.

Nyman, N. (2000, January/February). Using Monkey Test Tools: How to find bugs cost-effectively through random testing. *Software Testing and Quality Engineering, 2,* 18-23.

Pettichord, B. (2003, October 13-15). *Four Schools of Software Testing*. Paper presented at the Pacific Northwest Software Quality Conference, Portland, OR. http://www.io.com/~wazmo/papers/four_schools.pdf

Pressman, R. S. (2005). *Software Engineering: A Practitioner's Approach* (6th ed.). New York: McGraw-Hill.

Rainsberger, J. B. (2004). *JUnit Recipes: Practical Methods for Programmer Testing*. Greenwich, CT: Manning Publications Co.

Rosenthal, R. (1966). *Experimenter Effects in Behavioral Research*. New York: Appleton-Century Crofts.

Simmons, E. (2000). *When Will We be Done Testing? Software Defect Arrival Modeling Using the Weibull Distribution*. Paper presented at the Pacific Northwest Software Quality Conference, Portland, OR. http://www.pnsqc.org/proceedings/pnsqc00.pdf

Teasley, B. E., Leventhal, L. M., Mynatt, C. R., & Rohlman, D. S. (1994). Why Software Testing is Sometimes Ineffective: Two Applied Studies of Positive Test Strategy. *Journal of Applied Psychology, 79*(1), 142-155.