The Ongoing Revolution in Software Testing

Cem Kaner

Florida Institute of Technology

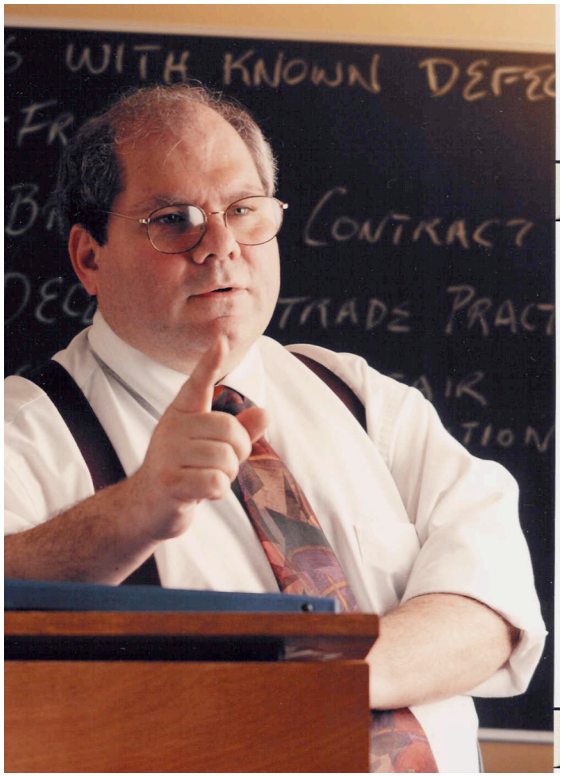
Software Testing & Performance Conference

Baltimore 12/8/2004

CENTER FOR SOFTWARE TESTING EDUCATION AND RESEARCH

About Cem Kaner

My current job titles are Professor of Software Engineering at the Florida Institute of Technology, and Research Fellow at Satisfice, Inc. I'm also an attorney, whose work focuses on same theme as the rest of my career: satisfaction and safety of software customers and workers.



I've worked as a programmer, tester, writer, teacher, user interface designer, software salesperson, organization development consultant, as a manager of user documentation, software testing, and software development, and as an attorney focusing on the law of software quality. These have provided many insights into relationships between computers, software, developers, and customers.

I'm the senior author of three books:

- ***Lessons Learned in Software Testing*** **(with James & Bret Pettichord)**
- ***Bad Software*** **(with David Pels)**
- ***Testing Computer Software*** **(with Jack Falk & Hung Quoc Nguyen).**

I studied Experimental Psychology for my Ph.D., with a dissertation on Psychophysics (essentially perceptual measurement). This field nurtured my interest in *human factors* (and thus the usability of computer systems) and in *measurement theory* (and thus, the development of valid software metrics.)

The Ongoing Revolution in Software Testing

Some Messages of this Talk

- **Testing involves an active, skilled, technical investigation. Competent testers are investigators--clever, sometimes mischievous researchers. I think that models of testing that don't portray testing this way are obsolete for most contexts.**
- **In much of the past 30 years, many leaders in the testing community have urged us to dumb our work down, make it more routine and then cost-reduce it. This often leads to serious inefficiency and weak testing. As a particular example, much regression testing should be moved to the unit test level or reconsidered. Automation is useful, but only if useful things are automated in efficiency-supporting ways.**
- **Suppose we thought of ourselves as who we are at our best:**
  - **active learners who find ways to dig up information about a product or process just as that information is needed.**
  - ***How would our attitudes about testing (and testers) change if we adopted this as our vision?***

Old Views

- **Many years ago, the software development community formed a model for the software testing effort. As I interacted with it from 1980 onward, the model included several "best practices" and other shared beliefs about the nature of testing.**

The testing community developed a culture around these shared beliefs.

Best Practices?

- Let's be clear about what we mean when we say, "Best Practice."
- A "best practice" is an idea that a consultant thinks he can sell to a lot of people. There is no assurance that this idea has ever succeeded in practice, and certainly no implication that it has been empirically tested and found superior (best) to competing ideas under general conditions.
- A "best practice" is a marketing concept, a way of positioning an idea. It is *not* a technical concept.

Old Views

- Much of the same old lore has stayed with us and is currently promoted as the proper guide to testing culture & practice. For example:
  - Look at ISEB's current syllabus for test practitioner certification:

    www1.bcs.org.uk/DocsRepository/00900/913/docs/practsyll.pdf
- Look at the IEEE's Software Engineering Body of Knowledge section on software testing

  www.swebok.org
- These, and many other presentations, could have been written almost as well in 1990 or even 1980.
- I think it's time to reject most of these ideas and move on.

Old Views

- I started writing Testing Computer Software to foster rebellion against some of these ideas.
  - 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.
- Oh, Pshaw!
- Of course you can do good testing under these circumstances.

Old Views

- **Even though TCS rejected several of the leading excuses, I adopted much of the rest of the received wisdom:**
  - ***Such as the idea that the sole purpose of testing is to find bugs***
- **Or wrote my critiques too gently for the average reader to realize that I thought the process was broken:**
  - ***Such as the idea that a test isn't meaningful unless you specify expected results***
  - ***Such as the idea that we should create detailed, procedural test documentation***
  - ***Such as the idea that we should develop the bulk of test materials fairly early in the project***
- **Or I stayed silent because I wasn't sufficiently confident of my conclusions:**
  - ***Such as heavy reliance on GUI regression test automation***
  - ***Such as the invalidity of most of the test-related metrics currently in use***

Old Views

- Over the past decade, since publishing TCS 2.0, I've become increasingly skeptical of traditional testing:
  - Too much of it doesn't scale to the ever-larger programs we are creating. It's great to lovingly handcraft and thoroughly document individual tests, but how much of this can you do when a cell phone comes with 2 million lines of code?
  - It ignores the problem that testing is such a huge task (infinite, actually) that we have to live by our wits in figuring out the right tradeoffs.
  - In glorying a failing proceduralism over skilled craft, it pushes bright people into other areas, creating a self-fulfilling prophesy of low skill in the field.
  - It fosters a toxic relationship between testers, programmers and project managers.
- In 1999, I decided to subject my views on testing to a fundamental reappraisal, and to drive toward training a new generation of test architects. To do this, I went back to school. . .

A Fresh Breeze

- In the 1990's, many members of the programming community finally decided to strike back in their own way at the ineffectiveness (and unpleasantness) of the test groups they worked with.
- They decided that if they couldn't rely on testers for good testing, they'd have to take back the responsibility for testing, themselves.
- The results were
  - Test-driven programming
  - Glass-box integration test tools, such as FIT
  - A variety of other open source test tool initiatives
  - A renewed distinction between programmer-testing and application-testing or customer-side testing.

Purpose of Testing?

- "Testing is the process of executing the code with the intent of finding errors"
- The purpose of testing is to find bugs.
  - A test that finds a bug is a success
  - A test that didn't find a bug was a waste of time

It's a wonderful definition

except that

we do a lot of testing for a lot of reasons

other than finding bugs.

Information Objectives

- Find defects that matter
- Maximize bug count
- Block premature product releases
- Help managers make ship / no-ship decisions
- Minimize technical support costs
- Assess conformance to specification
- Conform to regulations
- Minimize safety-related lawsuit risk
- Find safe scenarios for use of the product
- Assess quality
- Elucidate the design and prevent errors

Different objectives require different testing strategies and will yield different tests, different test documentation and different test results.

Let's fix the definition

Defining Testing

- **A technical**
  - We use technical means, including experimentation, logic, mathematics, models, tools (testing-support programs), and tools (measuring instruments, event generators, etc.)
- **investigation**
  - an organized and thorough search for information
  - this is an active process of inquiry. We ask hard questions (aka run hard test cases) and look carefully at the results
- **done to expose quality-related information**
  - see the slide on information objectives
- **about the product under test**

The Concept of Inertia

- **INERTIA: The resistance to change that we build into a project.**
- **The less inertia that we build into a project, the more responsive we (the development group) can be to stakeholder requests for change (design changes and bug fixes).**
  - **Intentional inertia:**
    - **Change control boards**
    - **User interface freezes**
  - **Process-induced inertia**
    - **Costs of change that are imposed by the development process**
      - **rewrite the specification**
      - **rewrite the tests**
      - **re-run all the tests**
- **Testers must realize that when they introduce heavyweight practices to a project, they increase the project's inertia and make it more resistant to improvement.**

Inertia #1
Document Manual Tests in Full Procedural Detail?

- **The claim is that manual tests should be documented in great procedural detail so that they can be handed to less experienced or less skilled testers, who will**
  - **(a) repeat the tests consistently, in the way they were intended,**
  - **(b) learn about test design from executing these tests, and**
  - **(c) learn the program from testing it, using these tests.**

I don't see any reason to believe that we will achieve any of these benefits. I think this is as close as we come to an industry worst practice.

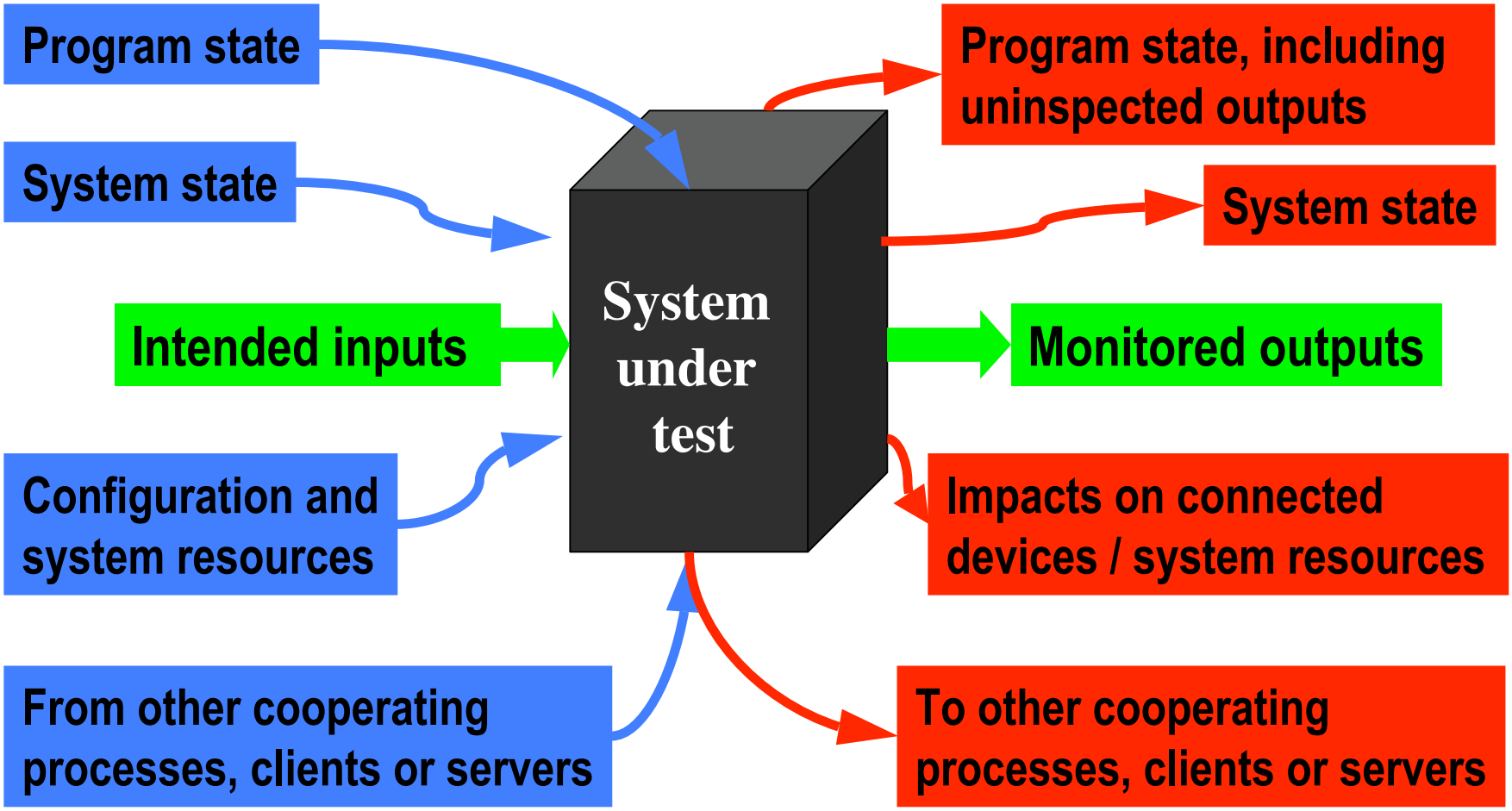
Inertia #2
Must we specify expected results?

- Glen Myers pushed this point very effectively
  - He illustrated it with a rant about the ineffectiveness of testing at IBM. Testers had created enormous test runs, but didn't know how to spot failures in their printouts. As a result, about 35% of the failures in the field could be traced back to failures that were actually exposed by tests run in the lab, but never recognized as failures by the testers.
- So, must we always specify expected results?
- Is it true, as we still hear at this conference, that test data that aren't tied back to expected results are meaningless?

Critical Problem

A test that is defined in terms of one expected result is undefined against the other types of results available from that test.

A program can fail in many ways



Based on notes from Doug Hoffman

Inertia #2
Must we specify expected results?

- A lot of testing actually involves working with the program in a certain way to figure out what it actually does, and to make sense of it -- whether what it does is appropriate or not.
- People (many, maybe most people) don't understand specifications and documentation just by reading them or drawing diagrams about them. You often learn about something by doing things with it.
- The idea of exploratory testing is that you recognize that you're going to learn while you test, that you're going to get more sophisticated as you learn, that you'll interpret your tests differently and design your tests differently as you learn more about the product, the market, the variety of uses of the product, the risks, the mistakes actually made by the humans who write the code. So you build time and enthusiasm for parallel research, test development and test execution.
- As you learn what you learn, while you test, you may or may not flag an individual result as noteworthy, worthy of reuse or re-execution.
- For many tests, by the time you come to understand what result you *should* expect, you've already gotten all the value you're going to get from that test.
- To demand the development of test case documentation first is to bar this type of technical learning.

Inertia #2
Must we specify expected results?

- One of the interesting things we find in load and performance testing is that we get functional errors--the program fails under load--from code that seemed to work fine when we ran functional tests.
- The failures often reflect long-sequence bugs, such as memory leaks, memory corruption, stack corruption, or other failures triggered by unexpected combinations of features or data.
- To find bugs like these intentionally, we can use a variety of high volume test automation techniques.
  - See Jeffrey Feldstein's talk at this conference
  - http://www.testingeducation.org/a/hvta.pdf
- A "problem" with these tests: we don't really have expected results. The results we would list as expected for each test have no relationship to the actual risks we're trying to mitigate.
  - In my consulting experience, I found that many test managers whose tests come in neat, well specified packages found it hard to even imagine high volume test automation or consider the idea of applying it to their situations.
  - BUT THESE TESTS FIND PROBLEMS THAT ARE HARD TO FIND ANY OTHER WAY

Inertia #3
Design Most Tests Early in Development?

- **Why would *anyone* want to spend most of their test design money early in development?**
  - **The earlier in the project, the less we know about how it can fail, and so the less accurately we can prioritize**

One of the core problems of testing is the infinity of possible tests.

Good test design involves selection of a tiny subset of these tests.

The better we understand the product and its risks, the more wisely we can pick those few tests.

Design Most Tests Early in Development?

"Test then code" is fundamentally different from test-first programming

Test then code ("proactive testing")	Test-first development
The *tester* creates many tests and then the *programmer* codes	The programmer creates 1 test, writes code, gets the code working, refactors, moves to next test
Primarily acceptance, or system-level tests	Primarily unit tests and low-level integration
Usual process inefficiencies and delays (code, then deliver build, then wait for test results, slow, costly feedback)	Near-zero delay, communication cost
Supports understanding of requirements	Supports exploratory development of architecture, requirements, & design

More on Test-Driven Development

- **Benefits of TDD to the project?**
  - **Provides a structure for working from examples, rather than from an abstraction.** ***(Supports a common learning / thinking style.)***
  - **Provides concrete communication with future maintainers.**
  - **Provides a unit-level regression-test suite** *(change detectors)*
    - *support for refactoring*
    - *support for maintenance*
  - **Makes bug finding / fixing more efficient**
    - **No roundtrip cost, compared to GUI automation and bug reporting.**
    - **No (or brief) delay in feedback loop compared to external tester loop**
  - **Provides support for experimenting with the language**

Good Tests Should be Reused as Regression Tests?

- Let's distinguish between the change-detectors at the code level and UI / System level regression tests
- Change detectors
  - writing these helped the TDD programmer think through the design & implementation
  - near-zero feedback delay and near-zero communication cost make these tests a strong support for refactoring
- System-level regression
  - provide no support for implementation / design
  - are run well after the code is put into a build that is released to testing (long feedback delay)
  - run by someone other than the programmer (feedback cost)

Good Tests Should be Reused as Regression Tests?

- Maintenance of UI / system-level tests is not free
  - change the design → discover the inconsistency → discover the problem is obsolescence of the test → change the test
- So 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 feedback does the test suite provide for the project?

Modern Approaches

- What can we achieve with unit testing?
  - We can eliminate the need for a broad class of boring, routine, inefficient system-level tests:
    - Hunt & Thomas, *Pragmatic Unit Testing,* focus on confirmatory tests, give the example of inserting a large value into a sorted list, and confirm that it appears at the end of the list.
    - We can test that method in many other ways, at the unit level.
      - Try a huge value -- Try a huge list
      - Try a maximum length list -- Try a max+1 length list
      - Try a null value -- Insert into a null list
      - Try a value of wrong type -- Try a tied value
      - Try a negative value -- Try a zero?
      - Try a value that should sort to the start of the list.
      - Exact middle of the list
      - Exercise every error case in the method

Modern Approaches

- Many automated UI tests are unit tests run at the system level.
  - If the programmers do thorough unit testing
    - Based on their own test design, or
    - Based on a code analyzer / test generator (like Agitator)
  - Then apart from a sanity-check sample at the system level, we don't have to do these as system tests.
  - Instead, we can focus on techniques that exercise the program more broadly and more interestingly
- Test-driven development groups can benefit from support from testers (pairing with programmers) via better TDD test design and better communication into the system test process

Modern Approaches

- There are a few hundred test techniques.
  - Some are focused on simple tests (narrow scope, could implement these test ideas in unit tests or few-unit integration tests).
  - Other tests assume a reasonably stable product (at the unit level) and let us get to harder-to-mitigate risks:
    - Scenario testing (http://www.kaner.com/pdfs/ScenarioSTQE.pdf)
    - High volume automation
    - State model based testing in presence of load (interrupts)
    - Many of the risks derived from failure mode and effects analysis
  - These may or may not automate efficiently, and they may or may not be best-automated with current commercial (as opposed to open source or not-yet-done) tools
  - We implement our tests as simply and close-to-the-underlying-code (maintenance benefits) as is effective and efficient for the given risks / tasks. Different companies will rationally select different tradeoffs.

Let's Draw Some Distinctions

- **Functional testing**

  Emphasis is on capability / benefits for the user.

  The skilled functional tester often gains a deep knowledge of the needs of customers and customer-supporting stakeholders.

- **Para-functional testing**

  Security, usability, accessibility, supportability, localizability, interoperability, installability, performance, scalability, etc.

  The customer / user is not an expert in these attributes but has great need of them.

  Effective testing will often require collaboration and mutual coaching between programmers and testers.

- ***We have to distinguish between testing for (e.g. contractual) acceptance and testing for evaluation of suitability, value, and utility. Both are important, but the automation tradeoffs apply differently.***

- **Preventative testing**

  TDD collaboration, specification evaluation, testability evaluation

  The tester provides support services to the programmers and designers, with the goals of preventing bugs or making them much easier to expose.

A Closing Shot at Metrics

- Very few companies have metrics programs today. But most companies have tried them. Doesn't that imply that most companies have abandoned their metrics programs?
- Why would they do that? *Lazy? Stupid? Unprofessional?*
- **Maybe the metrics programs added no value or negative value.**
- A key problem is that measurement influences behavior, and not always in the ways that you hope. (See Bob Austin's *Managing and Measuring Performance in Organizations*)
- Another key problem is that software engineering metrics are rarely validated. "Construct validity" (how do we know that this instrument measures that attribute?) almost never appears in the CS and SWE literature, nor do discussions on determining the nature of the attribute that we are trying to measure. As a result, our metrics often fail to measure what we assert they measure, and they are prime candidates for Austin-style side effects.
- Kaner / Bond at http://www.kaner.com/pdfs/metrics2004.pdf)

Summary

- Testing objectives vary, legitimately. Our testing strategy should be optimized for our specific project's objectives.
- "Best practices" can be toxic in your context. Do what makes sense, not what is well marketed.
- We test in the real world, we can provide competent services under challenging circumstances.
- Modern unit testing supports initial development of the program and its maintenance. It also makes it possible for the system tests to be run far more efficiently and effectively. But that coordination requires tester/programmer collaboration.
- UI level automation is high maintenance and must be designed for maintainability. Extensive GUI automation often creates serious inertia and may expose few bugs and little useful information.
- Automation below the UI level is often cheaper to implement, needs less maintenance and provides rapid feedback to the programmers.
- The value of a test lies in the information it provides. If the information value of a GUI-level test won't exceed its automation cost, you shouldn't automate it.
- Testing is investigation. As investigators, we must make the best use of limited time and resources to sample wisely from a huge population of potential tasks. Much of our investigation is exploratory--we learn more as we go, and we continually design tests to reflect our increasing knowledge. Some, and not all, of these tests will be profitably reusable.

A New Software Testing Society

- Association for Software Testing
- Pat Schroeder is President
- I edit the new Journal of the Association for Software Testing, a peer-reviewed journal that hopes to publish scholarly papers about practical testing matters.
  - (We're just forming the Editorial Board. To join JAST's Board, write kaner@kaner.com)
- www.AssociationForSoftwareTesting.org