The Ongoing Revolution in Software Testing

Cem Kaner, J.D., Ph.D.
Professor of Software Engineering
Florida Institute of Technology

Summary

My intent in this talk is to challenge an orthodoxy in testing, a set of commonly accepted assumptions about our mission, skills, and constraints, including plenty that seemed good to me when I published them in 1988, 1993 or 2001.

Surprisingly, some of the old notions lost popularity in the 1990's but came back under new marketing with the rise of eXtreme Programming.

I propose we embrace the idea that testing is an active, skilled technical investigation. Competent testers are investigators—clever, sometimes mischievous researchers—active learners who dig up information about a product or process just as that information is needed.

I think that

- views of testing that don't portray testing this way are obsolete and counterproductive for most contexts and
- educational resources for testing that don't foster these skills and activities are misdirected and misleading.

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 wrote *Testing Computer Software* to foster rebellion against some of these ideas and 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.
- *Excuse:* You can't test parafunctional aspects of a program (like performance, usability, security) because they are out of testers' scope.

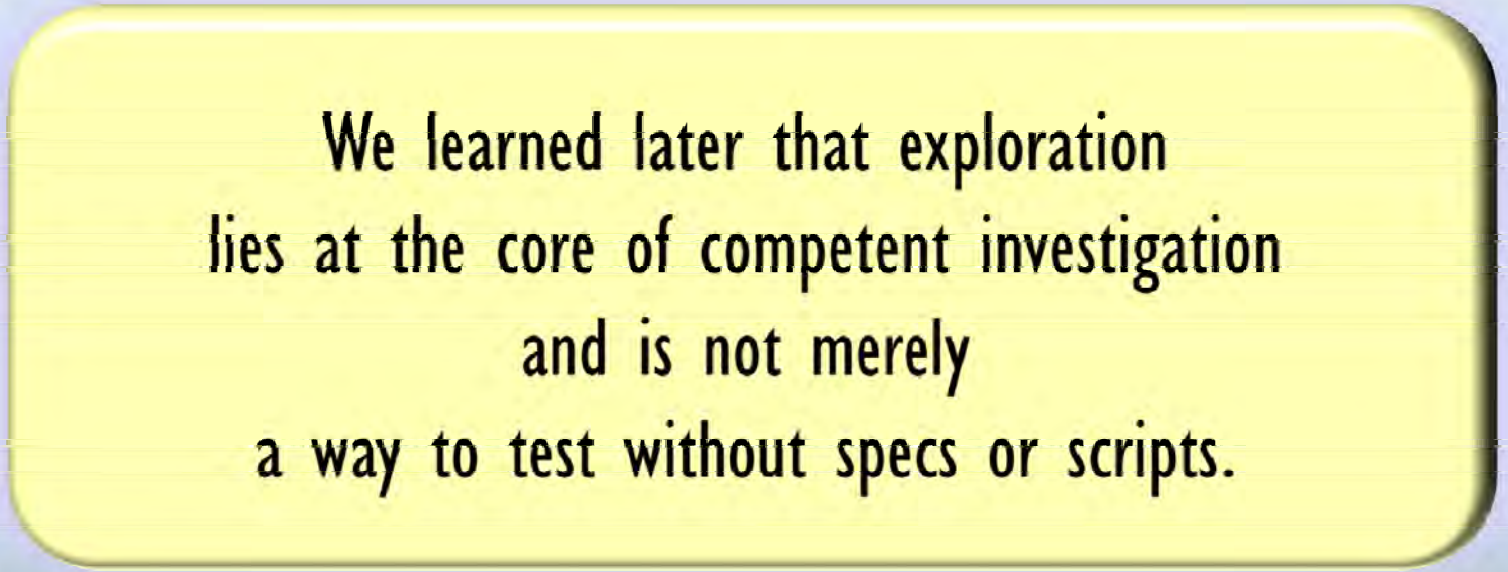
Oh, Pshaw!

Of course you can do good testing under these circumstances.

Oldies but goodies of the revolution

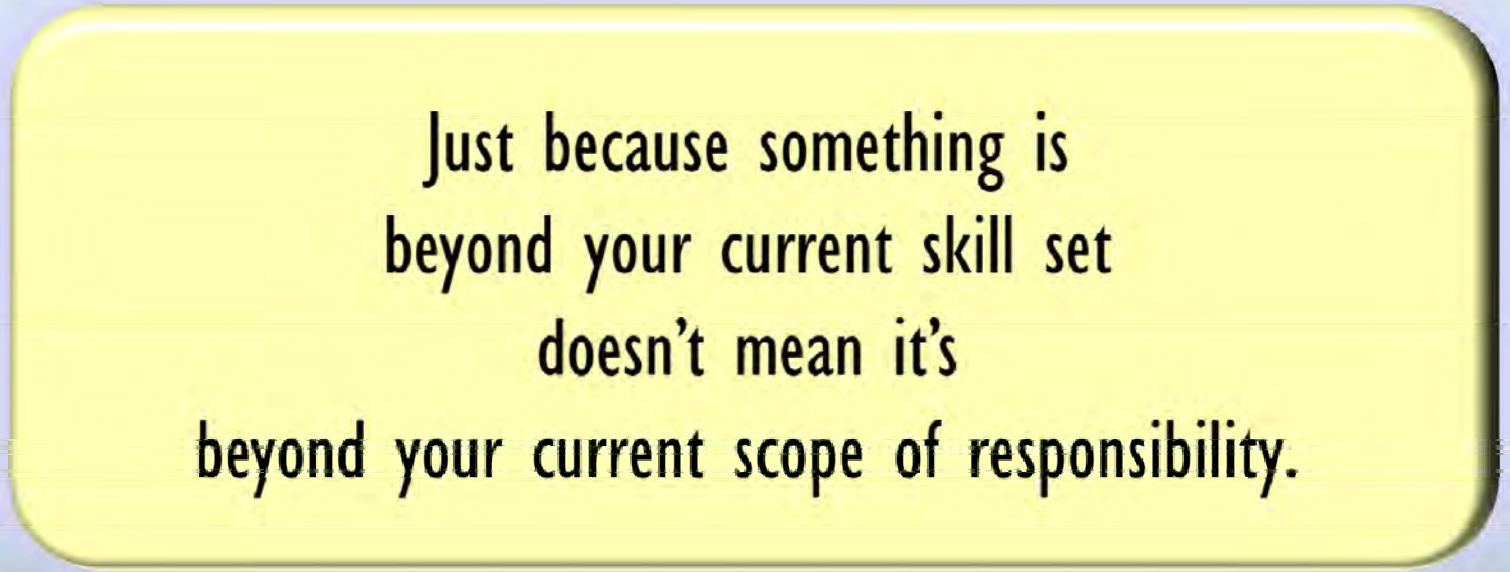
I wrote TCS to highlight what I saw as best practices (of the 1980's) in Silicon Valley, which conflicted with much received wisdom of the time:

- **Testers must be able to test well without authoritative (complete, trustworthy) specifications.** I coined the phrase, *exploratory testing*, to describe a survival skill.



Oldies but goodies of the revolution

- **Testing should address all areas of potential customer dissatisfaction, not just functional bugs.** Because matters of usability, performance, localizability, supportability, (these days, security) are critical factors in the acceptability of the product, test groups should become skilled at dealing with them.



Oldies but goodies of the revolution

- **It is neither uncommon nor unethical to defer (choose not to fix) known bugs.** The tester should research a bug or design weakness well enough to present that bug in its harshest light. You have done your job well if the project team understands the potential consequences of shipping with this bug when they choose to defer it.

Your task is to report the problem well,
not to decide whether to fix it.
If you want to ACT AS the project manager,
BECOME the project manager.

Oldies but goodies of the revolution

- **Testers are not the primary advocates of quality.** Testers are investigators. We help others understand the state of the product or process under test.
- Just because we gather the evidence doesn't mean we own the decisions or have any greater role in them than the other key stakeholders.

Testers do not assure quality.
We provide a quality assistance service
to a broader group of stakeholders
who take as much pride in their work as we do.

More oldies but goodies

- **The decision to automate a regression test is a matter of economics, not principle.**
  - It is profitable to automate a test (including paying the maintenance costs as the program evolves) if you would run the manual test so many times that the net cost of automation is less than manual execution.
  - Many manual tests are not worth automating because they provide information that we don't need to collect repeatedly.

5000 regression tests?
5000 tests the program has passed
over and over again.
Maybe we should try some other tests instead.

More oldies but goodies

- **Automation isn't just automated regression testing**
  - Other tests are worth automating—even if you only run them once—because the cost of doing them manually is too high and the information value of the test justifies the expense.
  - Automation and regression involve different considerations.

5000 automated regression tests?
Zzz Zzz Zzz
At least the program passes them quickly.
What could we find
with 5000 automated exploratory tests?

More oldies but goodies

- **Testers must be able to operate effectively within any software development lifecycle.** The project manager gets to decide what lifecycle is best for her project. That's why they call her "project manager."
  - Why do so many testers (and test consultants) push the waterfall model (including the V) so insistently?
    - phased development models push people to lock down decisions long before vital information is in, creating both bad decisions and resistance to later improvement.

You know less at the start of the project
than you will know at any later time.
Why make the most important decisions
at the time of greatest ignorance?

More oldies but goodies

- **Testers should design new tests throughout the project, even after feature freeze.**
  - All through the project, we will keep learning new things about the product, its market, its environment, and its risks.
  - For as long as we are learning new risks, we should be creating new tests.

The issue is not whether it is fair to the project team to add new tests late in the project.
The issue is whether the bugs those tests could find will impact the customer.

More oldies but goodies

- **We cannot measure the thoroughness of testing by computing simple coverage metrics or by creating at least one test per requirement or specification assertion.**
  - Thoroughness of testing means thoroughness of mitigation of risk.
  - Every different way that the program could fail creates a role for another test.

Complete testing means
the program is tested so thoroughly that
it is impossible
that there are undiscovered bugs.

Old Views: Mea Culpa

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**

I once said that
"A test that reveals a problem is a success. A test that did not reveal a problem was a waste of time."

Everyone makes mistakes...

Old Views: Mea Culpa

Even though TCS rejected several of the leading excuses, I wrote some 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**
- **Such as the idea that consistency of vocabulary is at all important in our field, or that knowledge of vocabulary is at all relevant to decided whether a tester is any good, compared to the urgent need for judgment and skill.**

Old Views: Mea Culpa

Even though TCS rejected several of the leading excuses, I stayed silent because I wasn't sufficiently confident of my conclusions:

- **Such as heavy reliance on GUI regression test automation**
- **Such as the chronic disconnects between IEEE standards and actual industry practice**
- **Such as the invalidity of most of the test-related metrics then (and now) in use**

Time for a Change

After publishing TCS 2.0, I became 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 (infinite) task that we have to live by our wits in figuring out the right tradeoffs.
- In glorifying a failing proceduralism over skilled craft, it pushes bright people into other areas, creating a self-fulfilling prophesy that our field attracts only low skill people.
- 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

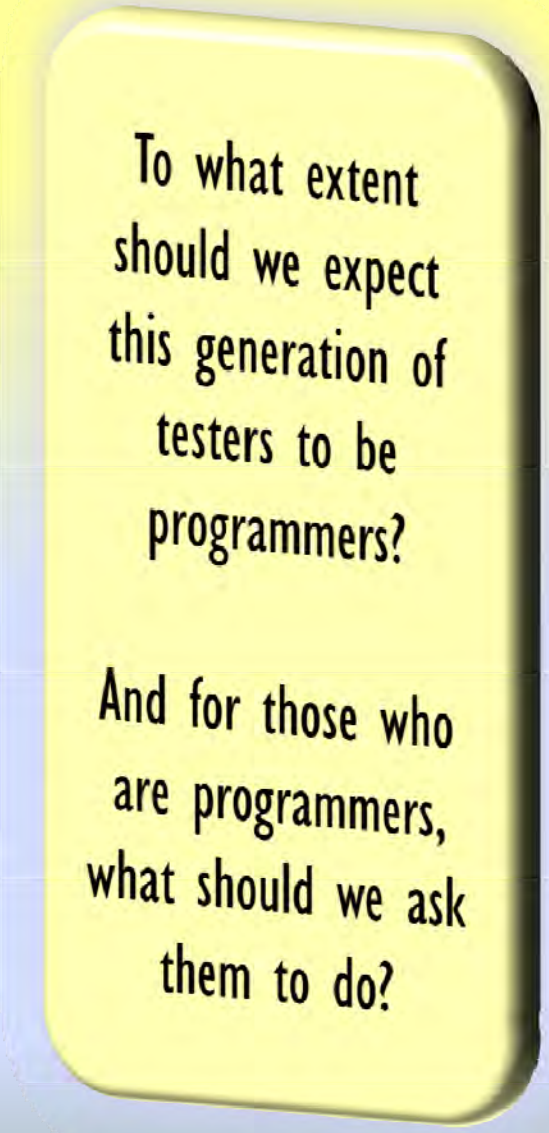
- Test-driven programming
- Glass-box integration test tools, such as FIT
- A variety of other open source test tool initiatives
- A renewed distinction between programmer-testing and application-testing or customer-side testing.

Sadly, in my view, even though the programmer-related testing ideas have been very valuable, the "agile" ideas about "customer" testing are disappointingly old (and stale).

The Test-Related Labor Market

Lots of advice that testers should work as programmers

- Unit and API test (independent or pair with programmers)
- Write GUI regression test suites
- Write performance tests
- Write test tools
- Write test code to drive devices or other systems
- Write non-regression tests that use technology to reach beyond what humans can do manually,
  - high volume (long sequence) testing
  - high precision testing
  - high diversity (directed search) testing



Our Labour Pool – data from 2004

- Nationally, CS enrolment is down 70% since 2001
- **90,000 new software development positions per year** (plus 29,000 support & hw positions).
- **60,000 computing B.Sc. grads**
  - (including computer engineers)
- 20,208 M.Sc. (many have B.Sc. already)
- 40,000 Associate degree (many go on to B.Sc.)
- Many of these are not from the top-ranked universities (2004 data):
  - DeVry Institute of Tech 3894 BSCS graduates
  - University of Phoenix 2552
  - American Intercontinental 1060
  - Strayer University 993

Labour Pool

U.S. tech job growth continues

U.S. IT employment continues on a growth path, rising 6% from a year ago to reach 3.68 million employed, according to the most-recent Bureau of Labor Statistics employment survey. IT unemployment was 2%, according to an average of the past four quarters of BLS data, including its most recent third-quarter results. That unemployment rate is down from 2.2% in 2006 and as high as 5.6% in the third quarter of 2003. The total IT workforce, employed and unemployed, also grew about 6% from a year ago. The unemployment rate in management and professional jobs overall was also 2.0%. The biggest job growth categories continue to be software engineers, computer scientists and systems analysts, and IS managers. Software engineers, the largest category, grew 8% from a year ago and make up a quarter of all IT jobs. *(InformationWeek 10/17/07)*

Our Labour Pool #2

- My understanding is since 2004:
  - open jobs have increased, while
  - CS enrolment has continued to significantly decrease.
- We appear to have touched bottom and might grow back significantly, but even if enrolment doubles in academic 2008-2009, those folks won't graduate until 2012.

Our Labour Pool #3

A CS degree is no guarantee of programming capability. I've visited schools around the country over the past two years.

- Several schools emphasize theory over programming skill (a senior professor at one school told me, "Few of our students can write a working100-line program when they graduate"). This is also widely perceived as a problem common to many CS graduates from India.
- Few CS or Software Engineering programmings emphasize (or even expose students) to soft skills (interviewing, context assessment, usability-oriented design, role playing, persuasive speaking and writing).
- Many courses in design and requirements analysis are essentially tutorials in patterns, UML, and creation of massive template-driven documentation.
- Many courses in software testing are broad and superficial.
- Another block of entrants into the field come from business schools, but many graduates with degrees in "Information Systems" have minimal education in software development or assessment.

Our Labour Pool #4

What I think this means…

- Of technically proficient graduates interested in testing, most seem to go to big publishers (Microsoft, Google) who aggressively recruit them.
- The IT community is unlikely to meet its needs for new testers with university graduate computer science majors who can write adequate code.

Over the next 5 years, few companies' new-hire testers will be appropriate for test-first programming, glass-box testing or serious test automation.

Labor Pool #5

- Will continue to include large portion of manual testers who have weak backgrounds in computing
  - 40,000 recent certifications by ISTQB
- The question will be how to hire and train the best people for a combination of:
  - Manual testing positions
  - GUI automation positions
  - Non-GUI (e.g. toolbuilder or HVAT) automation positions
  - Glass-box testing and test-first programming positions
- The proportions might shift over time, but the four roles (and in some companies, several other test-group roles) will continue.

Who should we hire?

For 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.

In my view, this often leads to serious inefficiency and weak testing.

Rather than

bringing testing down to a level that weak testers can do it (albeit it, weakly),

I think we should

Hire people with strong potential, and train them to do strong work.

Test groups should offer diverse, collaborating specialists

Test groups need people who understand

- the application under test,
- the technical environment in which it will run (and the associated risks),
- the market (and their expectations, demands, and support needs),
- the architecture and mechanics of tools to support the testing effort,
- and the underlying implementation of the code.

You cannot find all this in any one person. You can build a group of strikingly different people, encourage them to collaborate and cross-train, and assign them to project areas that need what they know.

http://www.kaner.com/pdfs/JobsRev6.pdf

Back to the Purpose / Nature of Testing

- Bug-hunting is a very important testing task
- But it's not the only one
- What else is there?

A better definition

Testing is an empirical, technical investigation conducted to provide quality-related information about a software product to a stakeholder

Defining Testing

Empirical

- We run experiments (tests). Code inspections are valuable, but they are not tests.

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

provide quality-related information

- see next slide (information objectives)

Information Objectives

Find important bugs, to get them fixed

Assess the quality of the product

Help managers make release decisions

Block premature product releases

Help predict and control costs of product support

Check interoperability with other products

Find safe scenarios for use of the product

Assess conformance to specifications

Certify the product meets a particular standard

Ensure the testing process meets accountability standards

Minimize the risk of safety-related lawsuits

Help clients improve product quality & testability

Help clients improve their processes

Evaluate the product for a third party

Different objectives drive you toward different:

- Testing techniques
- Project mgmt styles
- Results reporting methods
- Politics on the project

Testing is not manufacturing QC

Software testing is more like design evaluation than manufacturing quality control.

- A **manufacturing defect** appears in an individual instance of a product (like badly wired brakes in a car). It makes sense to look at every instance in the same ways (regression tests) because any one might fail in a given way, even if the one before and the one after did not.
- A **design defect** appears in every instance of the product. The challenge of design QC is to understand the full range of implications of the design, not to look for the same problem over and over.

By the way, Six Sigma is a manufacturing quality management methodology. The "six sigmas" are six standard deviations surrounding the mean of a probability distribution. I have never heard a rationale for applying this to software. (I've seen the enthusiasm, but not the mathematics.)

The Concept of Inertia

INERTIA: The resistance to change that we build into a project.

The less inertia we build into a project, the more responsive 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 imposed by the development process
    - rewrite the specification
    - rewrite the tests
    - re-run all the tests

When testers introduce heavyweight practices to a project, they increase the project's inertia and make it more resistant to improvement.

Inertia #1 -- Procedural documentation of manual tests

The claim is that manual tests should be documented in great procedural detail so they 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.

When you create a test script and pass it to an inexperienced tester, she might be able to follow the steps you intended, but she won't have the observational skills or insights that you would have if you were following the script instead. Scripts might create a sequence of actions but they don't create cognition.

Inertia #2 -- Expected results

Glen Myers pushed this point very effectively

- At IBM, testers created enormous test runs, but didn't know how to spot failures in their printouts. Result--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.
- I've seen this too.

HOWEVER: I have ***also*** seen cases in which testers missed bugs because they were too focused on verifying "expected" results to notice a failure the test had not been designed to address.

> You cannot specify all the results—all the behaviors and system/software/data changes—that can arise from a test.

Inertia #2 -- Expected results

- There is value in documenting the intent of a test, including results or behaviors to look for
  - but it is important to do so in a way that keeps the tester
  - thinking and
  - scanning for other results of the test
  - instead of viewing the testing goal as verification against what is written.
- A lot of testing involves working with the program to understand what it actually does-- 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.

Inertia #2 -- Expected results vs Exploration

- A lot of testing involves working with the program to understand what it actually does-- 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, and the mistakes actually made by the particular humans who write the code.
  - So you build time and enthusiasm for doing research, test development and test execution as parallel activities throughout the project.

Inertia #2 -- Expected results

- 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 development of the test case documentation
toward the start of the project
is to reject the benefit of the technical learning
that tells you what is (or is not) worth documenting.

Inertia #3 – Don't 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.

Don't 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
After 30 years, still rarely done.	Widely (not universally, but increasingly) adopted

More on Test-Driven Development

- 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 component library or language features

The value of unit testing

We can eliminate the need for a broad class of boring, routine, inefficient system-level tests:

- Hunt & Thomas, *Pragmatic Unit Testing,* ofetn emphasize confirmatory tests, such as giving the example of inserting a large value into a sorted list, and confirming 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

Unit tests and system tests #1

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 repeat these tests as system tests.
- Instead, we can focus on techniques that exercise the program more broadly and more interestingly

Unit tests & system tests #2: An example

Many testing books (including TCS 2) treat domain testing (boundary / equivalence analysis) as the primary system testing technique. To the extent that it teaches us to do risk-optimized stratified sampling whenever we deal with a large space of tests, domain testing offers powerful guidance.

But the specific technique—checking single variables and combinations at their edge values—is often handled well in unit and low-level integration tests. These are much more efficient than system tests.

If the programmers are actually testing this way, then system testers should focus on other risks and other techniques.

When other people are doing an honest and serious job of testing in their way, a system test group so jealous of its independence that it refuses to consider what has been done by others is bound to waste time repeating simple tests and thereby miss opportunities to try more complex tests focused on harder-to-assess risks.

Unit regression doesn't justify system regression

People who justifiably love unit testing preach that testers should invest heavily in system test automation too.

Change-detectors at the code level and UI / System level regression tests are very distinct.

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

- no support for implementation / design
- 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)

Cost/benefit the system regression tests

After they've been run a few times, a regression suite's tests have one thing in common: the program has passed them all.

In terms of information value, tests that offered new data and insights long ago, are now just a bunch of tired old tests in a convenient-to-reuse heap.

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?

Regression is not necessarily repetition

Procedural regression

- Do the same test over and over (reuse same tests each build)

Risk-focused regression

- Check for the same risks each build, but use different tests (e.g. combinations of previous tests)
- See http://www.testingeducation.org/BBST/BBSTRegressionTesting.html

Test automation isn't

Automated regression testing is not automated testing:

- we automate the test execution, and a simple comparison of expected and obtained results
- we don't automate the design or implementation of the test or the assessment of the mismatch of results (when there is one)

Like all other automated software testing
automated regression testing
is really
computer-assisted regression testing

What other computer-assistance would be valuable

- Tools to help create tests
- Tools to sort, summarize or evaluate test output or test results
- Tools (simulators) to help us predict results
- Tools to build models (e.g. state models) of the software, from which we can build tests and evaluate / interpret results
- Tools to vary inputs, generating a large number of similar (but not the same) tests on the same theme, at minimal cost for the variation
- Tools to capture test output in ways that make test result replication easier
- Tools to expose the API to the non-programmer subject matter expert, improving the maintainability of SME-designed tests
- Support tools for parafunctional tests (usability, performance, etc.)

High volume automated testing

Interesting finding in load and performance testing--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.

- 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

Testers may or may not work best in test groups

If you work in a test group, you probably get more testing training, more skilled criticism of your tests and reports, more attention to your test-related career path, and stronger moral support if you speak unwelcome truths to power.

If you work in an integrated development group, you probably get more insight into the development of the product, more skilled criticism of the impact of your work, more attention to your broad technical career path, more cross-training with programmers, and less respect if you know lots about the application or its risks but little about how to write code.

If you work in a marketing (customer-focused) group, you probably get more training in the application domain and in the evaluation of product acceptability and customer-oriented quality costs (such as support costs and lost sales), more attention to a management-directed career path, and more sympathy if you programmers belittle you for thinking more like a customer than a programmer.

Similarly, even if there is a cohesive test group, its character may depend on whether it reports to an executive focused on testing, support, marketing, programming, or something else.

There is no steady-state best place for a test group. Each choice has costs and benefits. The best choice might be a fundamental reorganization every two years to diversify the perspectives of the long-term staff and the people who work with them.

A Closing Shot at Common Testing 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. Only some of these tests will be profitably reusable.

About Cem Kaner

- Professor of Software Engineering, Florida Tech
- Research Fellow at Satisfice, Inc.

I've worked in all areas of product development (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.)

Senior author of three books:

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

My doctoral research on psychophysics (perceptual measurement) nurtured my interests in human factors (usable computer systems) and measurement theory.