The Past & Future of Software Testing:
How Many Lightbulbs *Does It Take* To Change A Tester?

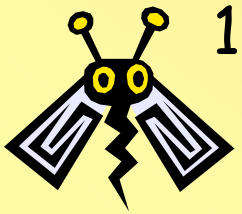
Cem Kaner, J.D., Ph.D.
kaner@kaner.com
www.kaner.com, www.testingeducation.org
at the

Software Testing Day

Tampere University of Technology

May 4, 2004

Research underlying these slides was partially supported by NSF 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 (NSF).



Old Truths

- 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.
- Beginning in 1983, I wrote Testing Computer Software to foster rebellion against some of these. And to support many others.

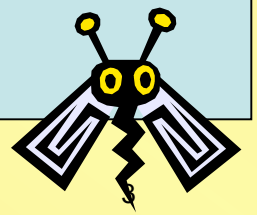
The testing community has developed a culture around these shared beliefs.

What should it take, for us to learn from our experiences?

Lightbulbs?

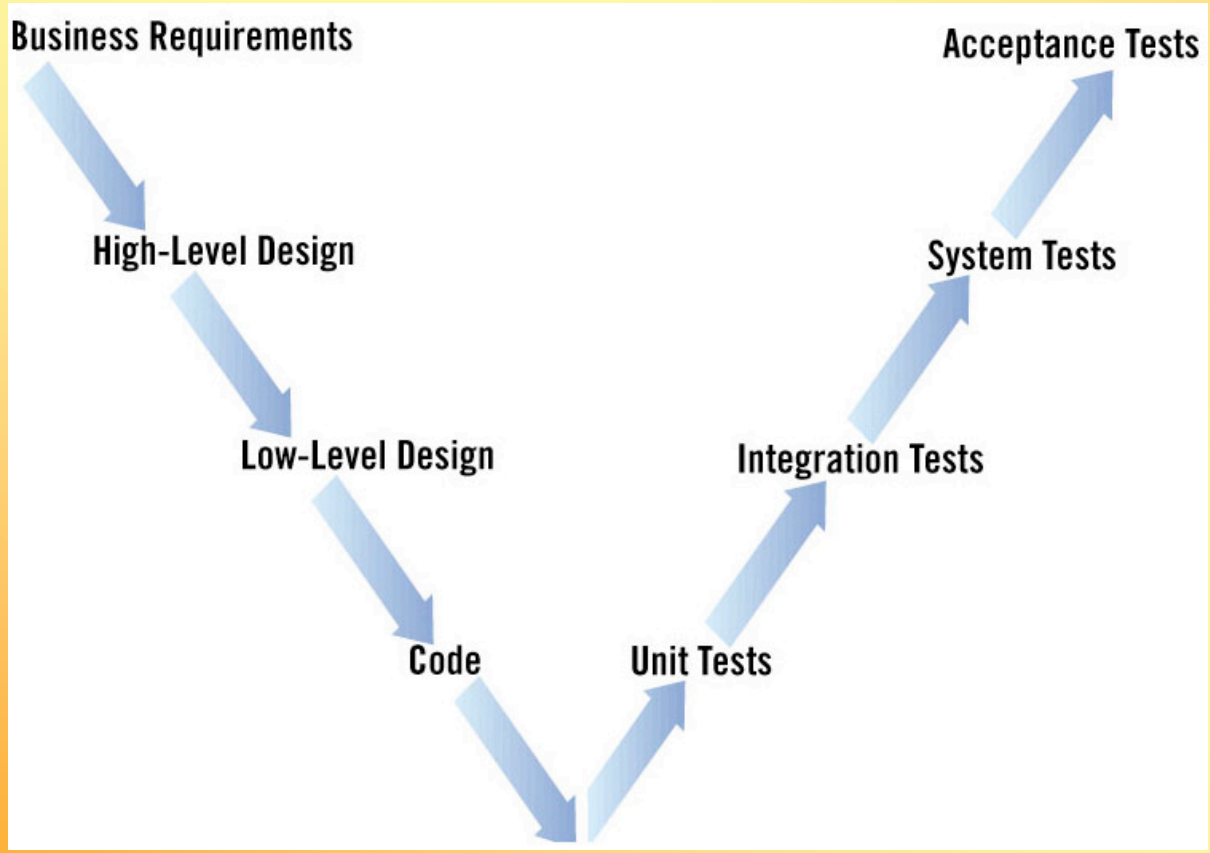
- Over the years, many
  - testing projects have failed
  - test managers have been fired
  - products have been successful despite corporate refusal to conform to testing "standards"
- And yet, most of the same old lore is still being repeated as the proper guide to testing culture and practice.

Don't think the old stuff is still with us?
Look at ISEB's current syllabus for test practitioner certification:
www1.bcs.org.uk/DocsRepository/00900/913/docs/practsyll.pdf
Or look at IEEE's SWEBOK, www.swebok.org
There are many other examples. . . .



We Should Advocate for Waterfall or V models?

- In these develop-in-stages lifecycle models, we try to finish each (stage / phase) before moving on to the next:

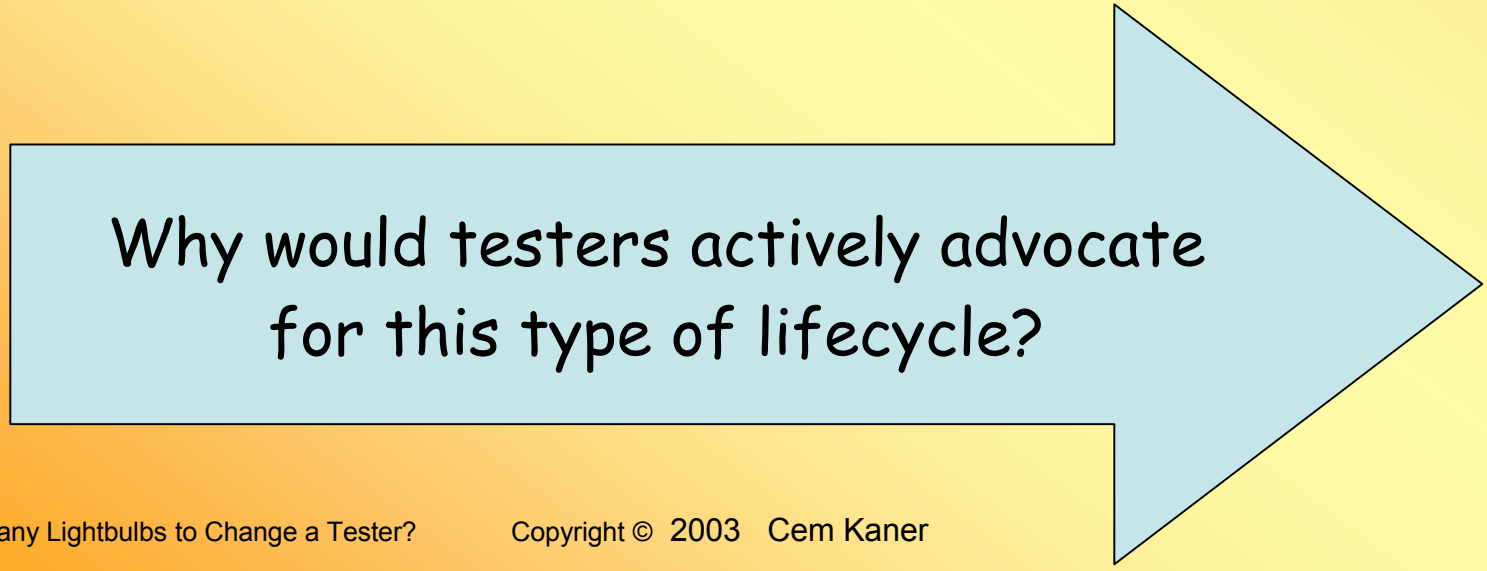


from

Robin Goldsmith, The Forgotten Phase, Software Development, July 2002, http://www.sdbestpractices.com/documents/s=8815/sdm0207e/0207e.htm

We Should Advocate for Waterfall or V models?

- These models are often harshly criticized as being inherently high risk.
  - 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 these risks.



We Should Advocate for Waterfall or V models?

- Think of the tradeoffs:

  -- Features -- Reliability -- Cost -- Time to Market

- In the waterfall (and V), we lock down the feature set early, do the design, write the code (and so spend most of the money that we ever will spend on it)
- So, toward the end of the project, what variables are still free?

Reliability versus **Time**

Sound familiar?

Why should we set ourselves up for this grief?

Iterative models are designed to change this tradeoff

Testers Should Refuse to Test if There is No Specification?

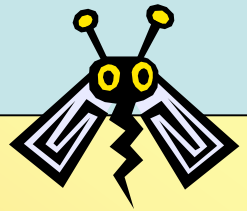
- This was one of the assertions that motivated me to start writing Testing Computer Software in 1983. (I disagreed with it.)
- It's still with us. I saw a leading consultant / author publicly give this advice to a pair of newly-promoted test managers a couple years ago.
- Let's face some realities
  - 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?*
  - A tester who refuses to proceed until the engineering process is changed is hijacking the management of the project.

Jump in front of the train enough times
and eventually you *will* get run over.

Drive Tests From the Specification?

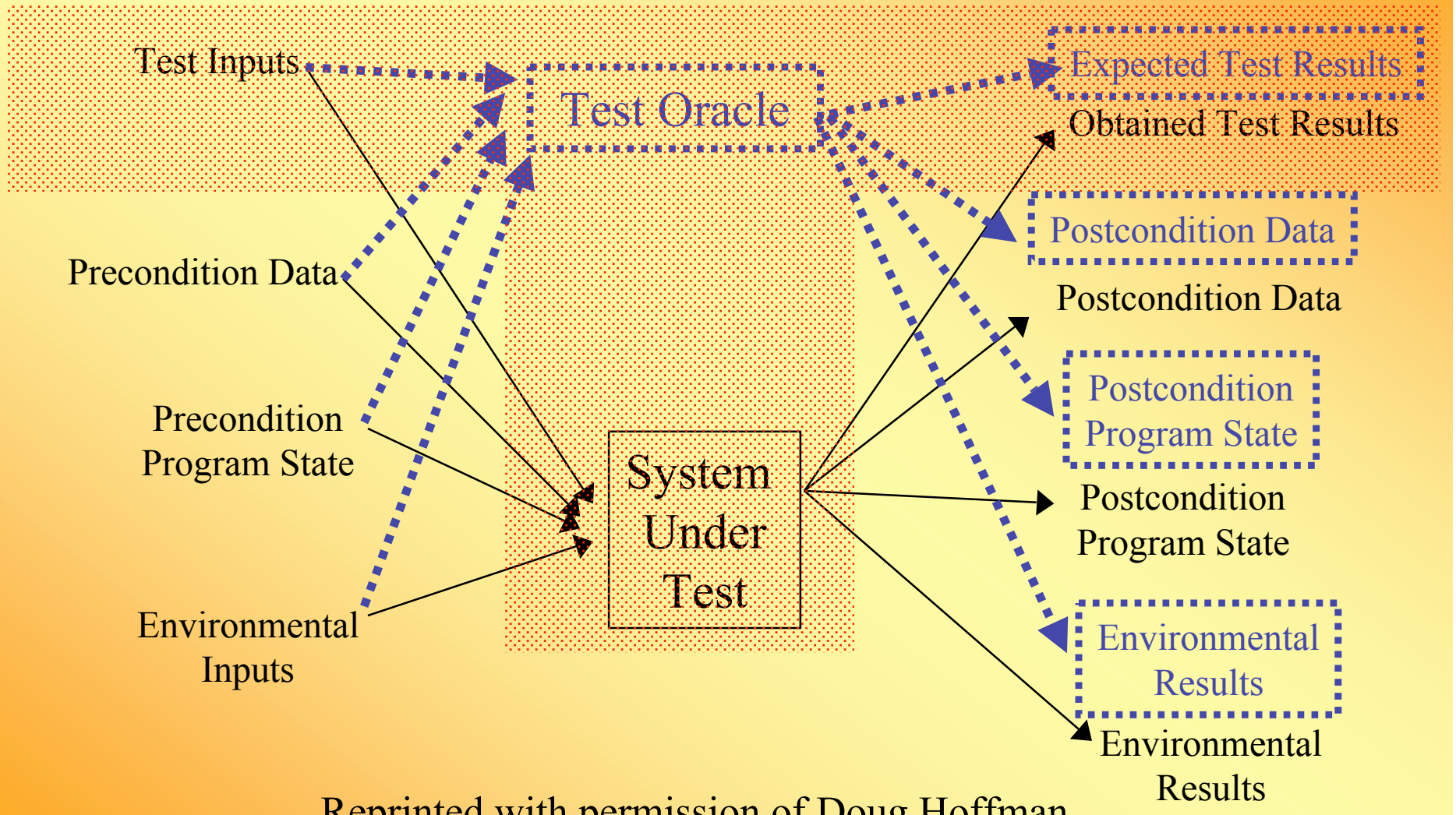
- It's easy to say that test cases should be based on documented characteristics of the program, for example on the requirements documents or the specifications.
- And there should be at least one thoroughly documented test for every requirement item or specification item.
  - Why only one test per item? Does one test each cover the spectrum of failure risks for these items? (No)
  - What about all the items in the program that can't be traced back to the requirements docs or the design specification?

A specification is **one** source of fallible guidance.
To the extent that the spec **limits** what you test,
it is a source of risk.



3

Expected Results are Only Part of the Story



Reprinted with permission of Doug Hoffman

A Test Without an Expected Result is Not a Test?

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

- High-volume automated tests may run until crash. The only "expected result" might be non-crash or conformance to other simple predictions. These techniques expose interesting problems that we do not know how to test for otherwise. The expected results have no relationship to the actual risks we are attempting to mitigate.
  - Are these improper tests?
  - Narrow visions of test design certainly steer us away from them.
  - ***But they expose critical bugs.***

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

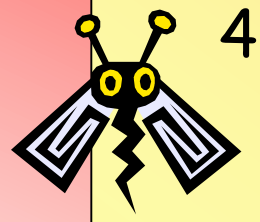
Test then code	Test-first development
The **tester** creates many tests and then the **programmer** codes	The programmer creates 1 test, writes code, gets the code working, refactors, moves to next test
Primarily acceptance, or system-level tests	Primarily unit tests and low-level integration
Usual process 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 & design

Good Engineering Requires Early Lockdown of the User Interface?

- How many times do we hear this nonsense from GUI regression test tool vendors and their shills?
- The user interface is there for communication with the user.
- Usability engineering is highly iterative

When we do beta testing
and discover,
as we always discover,
that the product confuses / annoys the user,

Do we really want to push a process that
will discourage developers
from making the improvements
we call for in our test results?



It's *Their* Job to Keep the User Interface Stable Not *Our* Job to Make Our Tests Maintainable

Mommy, mommy
those big, bad, nasty programmers
won't let me do my job!

It's THEIR fault I can't get anything done, Mommy!

- WHY do some testers think programmers will restructure their development process to make things convenient for testers?
- WHY would anyone expect programmers to show test code (and the tester) any respect if the code can't cope with simple changes?
- Change is inevitable. Deal with it.

Good Tests Should be Reused as Regression Tests?

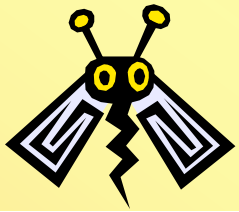
- Let's distinguish between the change-detectors at the code level and UI / System level regression tests
- Change detectors
  - writing these helped the TDD programmer think through the design & implementation
  - near-zero feedback delay and near-zero communication cost make these tests a strong support 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 does the test suite provide for the project?

The Concept of Inertia

- INERTIA: The resistance to change that we build into a project.
  - 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 advocate for late changes (aka bug fixes)
- What inertia do our processes induce in the project, and to what extent does our inertia block needed improvements?



5

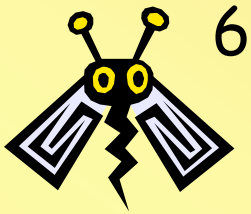
Testers are *The* Advocates of Quality?

How often have you heard statements like this?

- "*Somebody* has to look out for the user."
- "The project needs one group that really cares about quality."
- "Project managers can't afford to care about quality."

When you identify yourself as The Advocate For Quality you're saying that "We care about Quality *AND THEY DON'T.*

That prophesy gets self-fulfilling after a while.



Testers Should be Able to Veto a Product Release?

- The power to veto shipment is often described as the thing that distinguishes a “testing” group from a “quality assurance” group.
- There are serious problems with this “power”
  - It takes accountability away from the project manager who is making the development tradeoffs
  - It lets project managers (and others) play “release chicken”, setting the test group up to be the bad guys for holding 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

Testers Should be Able to Veto a Product Release?

- To ***achieve*** better quality, build a better product.
  - Better code, better user documentation, better training, better support.)
- To ***assure*** better quality,take over management of the groups that can make the product better.

Complaining about bad quality after the fact,
even beating people with a stick after the fact,
will not assure better quality.

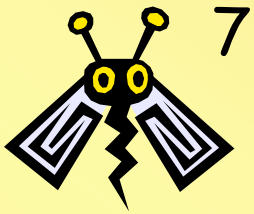
So What Do We Do?

- If we're not the quality assurance group, what are we?
  - Service providers.
  - We provide testing services.

Testing is a technical investigation of a product, an empirical search for quality-related information of value to a project's stakeholders.

Testers Should Work Without Knowledge of the Underlying Code?

- Why?
  - Protects testers from bias? *Huh?*
  - Gives us an excuse for hiring people who can't code?
- Maybe a better reason is discipline:
  - Rather than learning about the product from the code, the black box tester has to gather other classes of information that the programmer probably didn't consider.



So, Testers *SHOULD* Work Without Knowledge of the Code?

- **This is a practical question, not a question of principle**:
  - What does the project gain from a tester who focuses on customer benefits, configuration / compatibility, and coordination with other products?

contrasted with

  - 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?

Programmers Can't Catch Their Own Bugs?

- A programmer's **public bug rate** includes all bugs left in the code when she gives it to someone else (such as a tester.) Rates of one to three bugs per hundred statements are not unusual.
- A programmer's **private bug rate** includes all bugs she makes, including any she fixes before passing the program to testing.
- **Estimates of private bug rates:** 15-150 bugs per 100 statements (e.g. Beizer).

Therefore, programmers must be finding and fixing between 80% and 99.3% of their own bugs before their code goes into test.

- What does this tell us about our task?

We find bugs by looking into the programmer's (and her tools') blind spots.

- Merely repeating the types of tests that the programmers did won't yield more bugs.
- That's one of the reasons that an alternative approach is so valuable.

Testers Should Work Independently From Programmers?

Benefits of collaboration?

- **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.

  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.

- **Preventative testing (programmer support)**

Testers Should Work Independently From Programmers?

- **Preventative testing (programmer support)**

  Test-driven development (TDD) can benefit from support from testers (pairing with programmers)

  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

Our Job is to Find and Report Coding Errors, Not Design Errors?

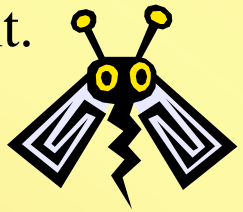
This is an old debate.

We can assess for many types of issue; few companies have specialists (e.g. UI expert), so we carry the problem.	VERSUS	Testers don't have skills to assess quality-characteristic problems (usability, security, accessibility, etc.).
We should build diverse test groups with knowledge in many relevant domains, including quality attributes		"Not my job" theory of testing – we should recruit and train for a narrow set of responsibilities
Testers are generalists, learning about all dimensions of quality problems		Testers are specialists, focusing on coding mistakes and playing a junior support role to programmers
Broad benefit-to-the-business vision of the role of testing		Programmer-centric vision of the role of testing

A key risk is the long-term de-skilling of the test group.

The IEEE Software Engineering Standards are a Basis for Sound Testing Practice?

- We see this assertion regularly, but
  - how many testers have ever read these standards? Have you? (*at PNSQC, only half were familiar with the IEEE software process standards and less than 1% advocated their use in their projects. How can this be an "industry standard"?)*
  - I think they have a strong bias toward heavyweight processes (massive paperwork, not much engineering).
  - I think they have a strong bias toward waterfall, which I see as high-risk engineering.
  - I think they have a strong bias toward a One True Way that doesn't vary with the project context.
  - On balance, I think Standard 829 has done more harm than good, and the other process standards have been largely irrelevant.



8

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

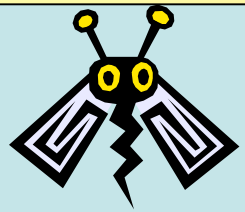
(c) learn the program from testing it, using these tests.

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.

Most Testers are Stupid or Non-Technical?

- Under this theory,
  - We attempt to reduce testing to a set of routine steps
    - document intensely so that one smart person can transmit The Great Thought to a bunch of lesser testers
    - automate intensely so that one smart person and a machine can replace those lesser testers
  - It makes sense to replace testers with test-first programmers
  - Test automation languages should be ultra-simple
  - Testers' time/work should be closely supervised with little room for independent judgment

This vision cannot scale to current product / development complexity



9

Ultimately We Face a Conflict of Visions

QA / QC	Active Learner
We are trying to develop a simple, easily taught, easily supervised, control process.	We are constantly trying to learn new things about the product (and how it can fail)
We want our infinite set of tests to feel finite. We use progress measures (e.g. "coverage") to gauge minimally-sufficient testing.	Breadth-first, then learning-guided depth, until we run out of ideas or time. We rely on cognitive triggers and peers to keep the flow of ideas steady (until we run out)
We support a process model or a narrow constituency (programmers *or* in-house end users *or* ...)	We support and influence a broad group of stakeholders; communication skills are vital.
Tools help us replace manual labor with automated labor	Tools help us learn new things

We Should Not Do Exploratory Testing?

- Exploratory testing involves simultaneously
  - executing tests
  - learning about the program, its market, its risks
  - designing new tests based on what we have just learned
- These tests can be automated or manual, whatever will teach us more about what we're investigating in the moment

Exploratory testing is the tool of
the active learner,
the technical investigator,

Rather than the QC automaton.

We Should Not Do Exploratory Testing?

- Everyone explores to some degree
  - follow-up testing (try to replicate or extent) a just-found bug
  - regression testing an allegedly-fixed bug
- Some tests provide all their value the first time you use them
  - Many scenario tests inform the tester of the design and provide little new information (compared to new scenarios) on reuse
- Straw man objections
  - Exploratory testing should only be done by experts (SWEBOK)
  - All testing should be automated, and if impossible to automate, should be made into a precise routine as if it were automated (Crispin)
  - All tests should be captured and turned into regression tests

We Should Not Do Exploratory Testing?

- Another way to avoid exploratory testing (and the vision of a skilled technical investigator) is to find a way to recharacterize exploration as something easily turned into a routine:
  - Exploration is "really" based on memorization of past failures: we could get the same benefit from a fault catalog, a failure catalog, or application of PSP.
  - Exploration is "really" based on a stereotyped set of attacks: maybe we can build a program that will generate these attacks.

Reference materials and improved tools certainly help the explorer explore, but ultimately, exploratory testing is about discovery of things we don't know about the program and don't yet have a routine, cost-effective system for exposing (that can work in the present project's context).

What's With All This Outsourcing?

- Peter Drucker, *Managing in the Next Society*, stresses that we should manufacture remotely but provide services locally. The local service provider is:
  - more readily available, more responsive, and more able to understand what is needed
- **So why are companies so willing to obtain their software development services from halfway around the world?**

If we adopt development processes that (at great cost) push all communication to paper, demand early decisions and make late changes difficult and expensive, what benefit is left to the local service provider? If we spend the money to create the formalistic infrastructure that we would need for outsourcing, we may as well do the work where it is cheaper, because we have squandered our local advantages.