

# How Many Lightbulbs Does It Take To Change A Tester?

Cem Kaner

Pacific Northwest Software Quality Conference

October 14, 2003

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).

Oh yeah? Well how many **TESTERS**  
does it take to change a **LIGHTBULB?**



# The Message of This Talk

- Testing involves an active, skilled, technical investigation.
- Competent testers are investigators -- clever, sometimes mischievous, researchers.
- In much of the past 20 years, many leaders in testing community have urged us to dumb our work down, make it more routine and then cost-reduce it.
- Time and again, I think we've seen unsatisfying results of this approach.
- Let's think 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 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.
- In 1983, I started writing *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.

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](http://www1.bcs.org.uk/DocsRepository/00900/913/docs/practsyll.pdf)  
Or look at IEEE's SWEBOK, [www.swebok.org](http://www.swebok.org)  
There are many other examples. . . .

***How many testers does it take  
to change a lightbulb?***



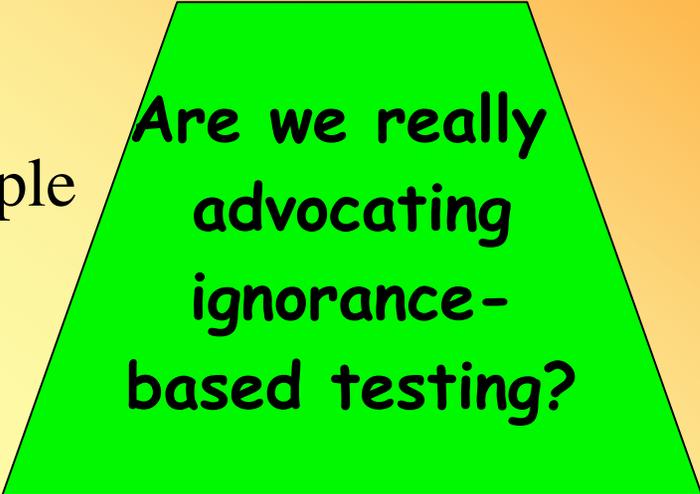
**Change?  
You want us to change?**

**We don't know how to do that . . .**

**But if you'd like, we could  
break some more bulbs for you...  
We're pretty good at *that*.**

# 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.



**Are we really  
advocating  
ignorance-  
based testing?**

*How many testers does it take  
to change a lightbulb?*



**What do you mean, change the lightbulb?**

**We're BLACK BULB TESTERS!**

**WE ALWAYS WORK IN THE DARK!**

# 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?

***How many testers does it take  
to change a lightbulb?***



**Just one, but**

**we've already spent our hardware  
budget for this quarter**

**so it will have to wait for  
a few months**

# 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?

- This often reflects fear and frustration
  - fear of being biased by programmers, away from good tests
  - fear of being distracted by programmers' agendas
  - fear of having programmers' work delegated to them
  - frustration at dealing with adversarial programmers

*How many testers does it take  
to change a lightbulb?*



**You want me to do WHAT?**

**DO YOU REALIZE HOW TIGHT MY SCHEDULE IS?**

**You're just trying add to my workload  
to distract me from  
finding more bugs.**

# 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

**Moral of the story:  
Even if it passes unit testing  
We can break it in system testing.**



**The joke:**

A development team finished a project and decided to take a vacation together to celebrate. But when they got on the train, the testers had only one ticket (for six testers). The programmers ridiculed them but the testers said, "Wait and see."

When the conductor reached the next car of the train, the testers all piled into the mens' bathroom. When the conductor reached the developers' car and asked for tickets, the testers passed their ticket out, under the bathroom door. The conductor accepted it and five testers got to ride for free.

On the way back, the programmers decided to buy a single collective ticket, like the testers had. But this time, the testers had no ticket. The programmers ridiculed them again, but the testers were confident as usual.

This time, when the conductor came near, the programmers all piled into the mens' room. Then the testers took over the ladies' room. But just before the conductor got to the car, one of the testers came out of the ladies' room, knocked on the door of the mens' room, and said, "Tickets please."

# The Purpose of Testing is to Find Bugs?

- Is this really true?

**This must be true.**

**Myers said it.**

**Even Kaner, Falk & Nguyen said it.**

***Surely, they know the right answer.***

- If a test case doesn't find a bug, does that really make it a failure? Or a waste of time?

***How many testers does it take  
to change a lightbulb?***



**They don't let us fix the bulbs,  
We just get to report when they're broken**

# The Purpose of Testing is to Find Bugs? Marick's Counter-Example



## *Testing Group 1*

	Found pre-release
Function A	100
Function B	0
Function C	0
Function D	0
Function E	0
Total	100

## *Testing Group 2*

Function A	50
Function B	6
Function C	6
Function D	6
Function E	6
Total	74

From Marick,  
*Classic Testing  
Mistakes*

Two groups test the same program.

- The functions are equally important
- The bugs are equally significant

*This is artificial, but it sets up a simple context for a discussion of tradeoffs.*

# The Purpose of Testing is to Find Bugs?

## Marick's Counter-Example

- The first group did narrower testing but found more bugs
- The second group did broader testing but found fewer bugs
- Marick preferred the second group, largely because
  - “The testing team will serve the project manager better if it concentrates first on providing estimates of product bugginess (reducing uncertainty)”

**Could it be that the *real* role of the test team is to help the project manager make difficult decisions?**

# The Purpose of Testing is to Find Bugs?

- But what if scheduling and release decisions are based on other criteria, not the test group reports?
- The primary benefit from Test groups varies from company to company:
  - Expose bugs
  - Block premature product releases
  - Build good-process record in case of lawsuits
  - Drive down tech support costs
  - Help ensure regulatory compliance
  - Assess conformance to contractual specification
  - Help managers make tough decisions

# The Purpose of Testing is to Find Bugs?

Perhaps we should think of the test group as technical investigators who dig up information that the company needs.

The key mistake would be to create a mismatch between the services you provide and those the company relies on you to provide

In practice, the key investigative goal / result is often  
(surprise!)

**FINDING BUGS**



# 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.	V E R S U S	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.

*How many testers does it take  
to change a lightbulb?*



**Leave it alone.**

**We're doing configuration testing.**

# 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.”



*How many testers does it take  
to change a lightbulb?*



**Four**

**One to change the bulb**

**And three to complain about  
the crummy equipment  
they have to put up with**

# Testers and Programmers have Conflicting Interests?

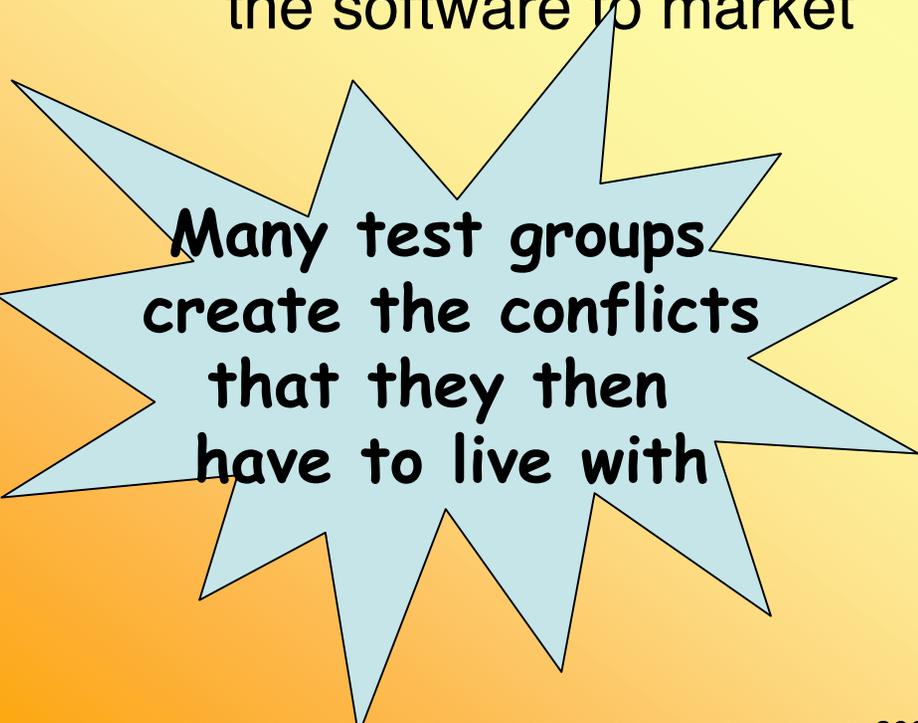
- Supposedly,
  - The test group’s job is to get the software “right” whereas
  - The programmers’ and project managers’ job is to get the software to market

The project manager’s challenge is to ship:

- The right features
- at the right reliability level
- at the right cost
- on the right schedule

These are tough tradeoffs.

If your view of them is adversarial, you might not have much influence over the decisions.



**Many test groups  
create the conflicts  
that they then  
have to live with**

*How many testers does it take  
to change a lightbulb?*



**YOU want US to CHANGE something?**

**You say it will make us test more efficiently?**

**WHAT DO YOU KNOW ABOUT TESTING?**

**You don't want us to be more efficient.  
You just want us to find fewer bugs.**

**Have you no quality standards?  
No shame?**

# 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

*How many testers does it take  
to change a lightbulb?*



**I don't know.**

**We might have to change  
all the bulbs  
to preserve  
process consistency.**

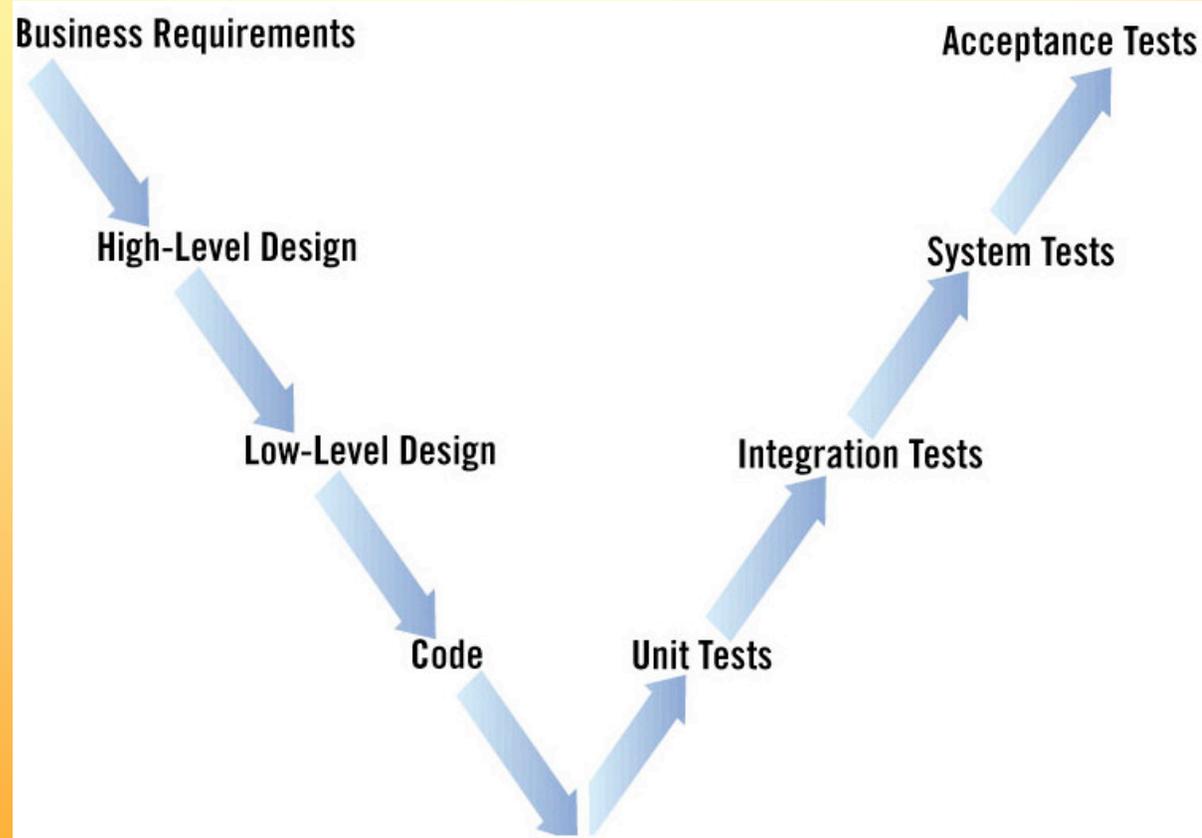
# Testers Should be Able to Veto a Product Release?

- 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.

# 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. For example, 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 against these risks.

Why would testers actively advocate  
for this type of lifecycle?

# 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 Cost**

**Sound familiar?**

**Why should we set ourselves up for this grief?**

**Iterative models are designed to change this tradeoff**

# 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

<b>Test then code</b>	<b>Test-first development</b>
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 skills?
- 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?

*How many testers does it take  
to change a lightbulb?*



**Three.**

**One to report the problem.**

**And two to kick the ladder out  
from under the programmer  
who tries to fix it.**

# 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?

***How many testers does it take  
to change a lightbulb?***



**I don't know.**

**Do you think we'll be allowed  
to make this change  
by the  
Change Control Board?**

# 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.

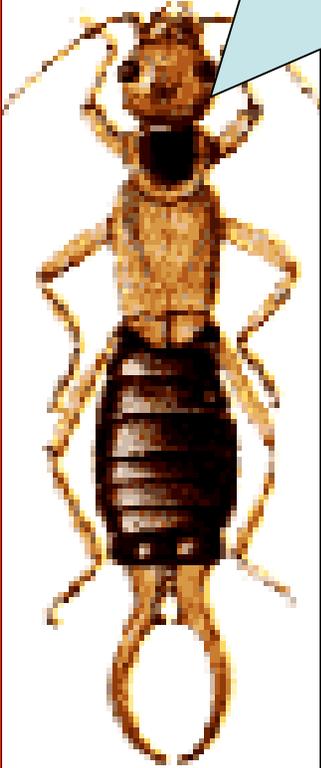
## Testers Should Refuse to Test if There is No Specification?

- 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.

# 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?
  - 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.



According to the IEEE (Std 829), it takes 47:

- 1 to write the test incident report
- 1 to trace bulb test back to requirement specs
- 1 to request design specification for lightbulbs
- 1 test manager to threaten to stop work until all lightbulb specifications are received
- 1 each to review the lightbulb user, installation, and operations guides
- 17 to define the test items for lightbulbs, including:
  - Bulb test project introduction
  - Bulb features not to be tested
  - Bulb test approach
  - Bulb test cases
    - Full procedural details for all test cases
  - Etc. .
- Etc.

***Plus 3 to brief accounting and sales on how to charge the client for all this paperwork***

## 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.

## 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.

*How many testers does it take  
to change a lightbulb?*

None in our shop.

**That's such a routine procedure . . .**

**Management will outsource it  
to testers in India.**



# All Failures Should Be Reported / Stored in the Bug Tracking Database?

- Strong negative reaction among testers at Quality Week to the idea that in XP, we might write bug reports on 5x7 cards

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

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

**What additional capabilities from the new system?**

**Do the new capabilities interfere with  
achievement of the core objective?**

*How many testers does it take  
to change a lightbulb?*



**What do you mean, CHANGE THE BULB?**

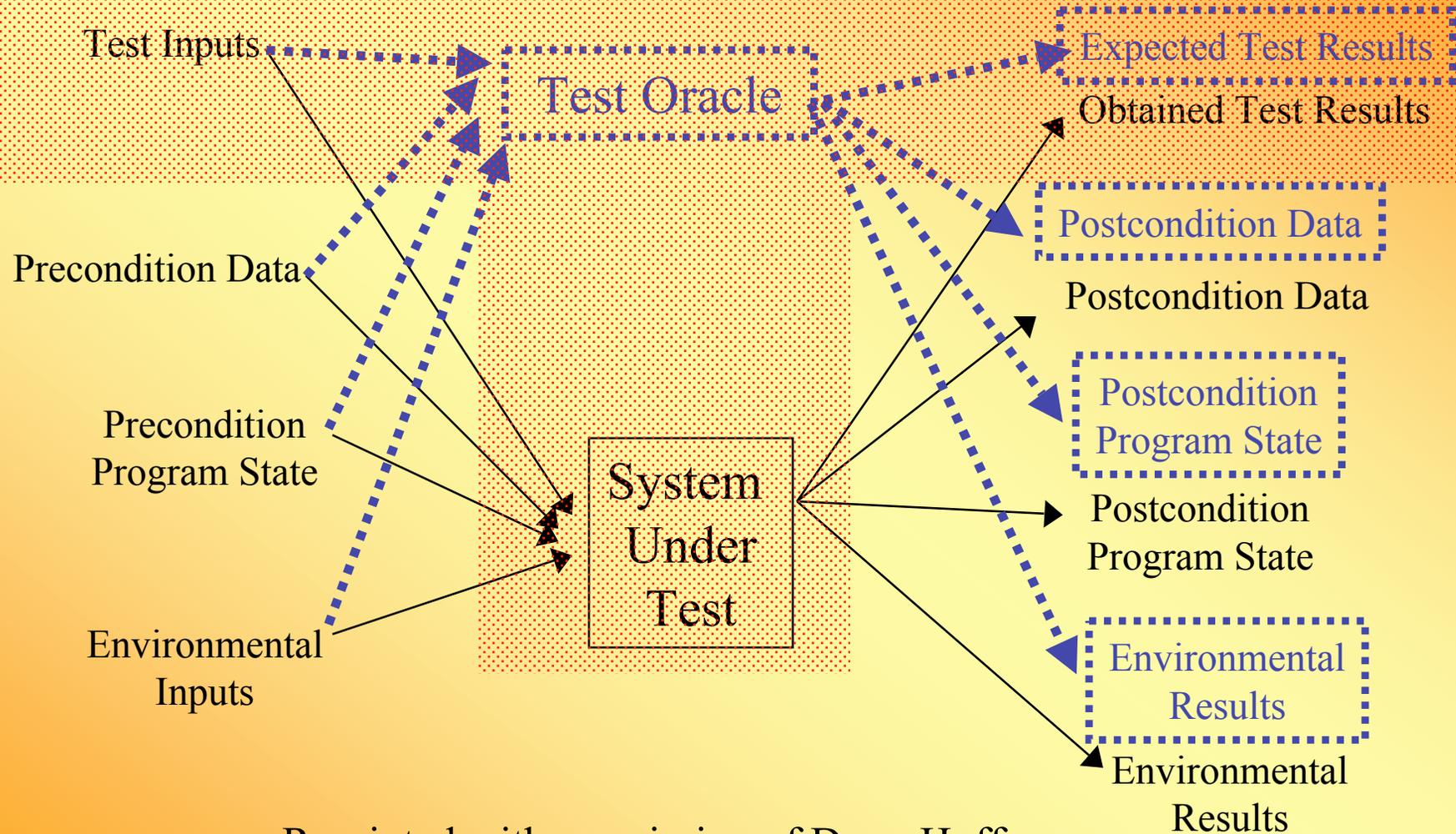
**That's a HARDWARE failure.**

**We're not even supposed to REPORT  
hardware failures!**

# A Test Without an Expected Result is Not a Test?

- Testers often explore, using heuristic rules to evaluate the program against “reasonable expectations”
  - When the program is lightly or inaccurately specified, the heuristics are primary guides. We don’t have authoritatively-derived expected results.
- High-volume automated tests may run until crash. The only “expected result” is non-crash. These expose interesting problems; of course they are tests.
- A test that is defined in terms of one expected result is undefined against the other types of results available from that test.

# Expected Results are Only Part of the Story



Reprinted with permission of Doug Hoffman

# 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.

***How many testers does it take  
to change a lightbulb?***



**There's nothing about lightbulbs in  
the requirements documents.**

**I don't think we have to report this  
one to the client.**

**OK, so let's move on to the next problem report....**

# 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.**

**How many testers does it take  
to change a lightbulb?**

**Two**

**One inserts a super-high-watt bulb**

**The other (his boss)  
explains to Management  
that boundary tests are  
entirely appropriate  
even if they do sometimes  
burn out the light fixtures.**



# 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).

**Moral of the story:**  
**Subject matter expertise enables testers to take advantage of differences among equivalent representations**



**The joke:**

Joe is dying. His three friends are with him at the hospital and he begs them: "They say you can't take it with you when you go, but I want to try. When I die, I want you to sell my assets and bring the money with you, 1/3 each. When they cremate me, burn my money with me. His friends agreed, eventually. Reluctantly.

Joe died. His friends sold his property, came to the funeral, tossed in 3 big green garbage bags and went to the bar for a drink.

Joe's doctor, Sam, spoke first: "I kept \$5000 for the cancer fund. No one should die like Joe, and he'll never miss it."

Father Pat spoke next: "I'm glad someone else was the first to admit it. I kept \$10,000. Not for me, but for the Church. He made a pledge to the school of \$10,000 and hadn't paid it before he died. No man should face his maker with an unpaid debt like that.

David, Joe's third friend, was a software tester. He looked at Sam and Father Pat with sadness and compassion. Then he explained: "I can't believe you did that. I never would have done something like that. I put in a check for the entire amount."

# All Tests Should Be Automated?

- Why automate?
  - Sometimes the automation lets us explore risks that we can't reach by hand (e.g. timing precision)
  - Sometimes the automation lets us manage volumes that we can't reach by hand (e.g. sift through masses of data)
  - Sometimes the automation is the natural way to run the test (e.g. SQL code to test contents of database)
  - Sometimes the automation merely captures what we are already doing by hand
    - Cost / benefit tradeoff – automation is not free
    - Cost / benefit tradeoff – one-shot tests have value

*How many testers does it take  
to change a lightbulb?*



**Didn't we agree to automate  
all these recurring tasks?**

**So how do we change this bulb  
with Mercury?**

# Automated Execution: Is this Really Automation?

- Analyze product -- human
- Design test -- human
- Run test 1st time -- human
- Evaluate results -- human
- Report 1st bug -- human
- Save code -- human
- Save result -- human
- Document test -- human
- Re-run the test -- MACHINE
- Evaluate result -- machine plus human if there's any mismatch
- Maintain result -- human

**Woo-hoo! We really get the machine to do a *whole lot* of our work!**

(Maybe, but not this way.)

***How many testers does it take  
to change a lightbulb?***



**17**

**1 to fix it  
and 16**

**to rewire the building to  
automatically flip all the lightswitches  
for automated regression testing.**

**(They call this Bulb Verification Testing)**

**(The electrician thinks it will turn into a smoke test)**

## 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?

# 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**

***How many testers does it take  
to change a lightbulb?***



**1 tester  
to change the bulb**

**Plus a tech support person  
to show the tester  
how to plug  
the lamp back in.**

# 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 <i>or</i> in-house end users <i>or</i> ...)	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