# Software Testing as a Social Science

Cem Kaner

Florida Institute of Technology

Siena, Italy

IFIP Working Group 10.4

Software Dependability

July, 2004

CENTER FOR SOFTWARE TESTING EDUCATION AND RESEARCH

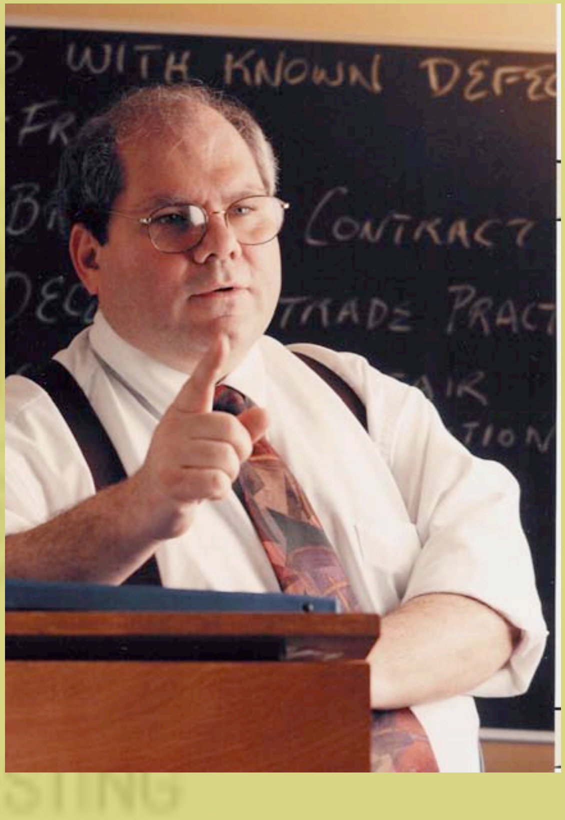
# About Cem Kaner

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

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

I'm  the senior author of three books:

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



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

# Welcome to Testing

*What is Testing?*

*A technical investigation done to expose quality-related information about the product under test*

# Defining Testing

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

# Information Objectives

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

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

# Software Testing as a Social Science

- The social sciences study humans in society.
- Software testers operate on technical products (computer programs--creations dominated by human thought).
- And testers use technical products (measuring instruments, test tools, etc.) and might even create technical products.
- But their core questions are often more human-focused than technological. For example
  - The tester's most common task is to find other people's mistakes, simplify the method of discovery, and persuasively present a report of the discovery to someone else, who will apply an economic model to determine whether to fix it.
  - The (probably) hardest to learn and hardest to teach testing skill is stretching your imagination to find new potential ways a program could fail. This skill is the essence of test design.
  - Measures of test-related variables are often of human performance variables

# Software Testing as a Social Science

- The talk today explores two questions:
  - What are some of the human issues in the *core* work of software testers?
  - How does thinking about these issues give us insight into the possibility of testing for human-error-related undependability?

# Let's Start with a Myth
# Programmers Can't Find Their Own Bugs (?)

- **Independent testing is often "justified" on the basis of an assertion that testers can't find their own bugs, just like writers can't (allegedly) find faults in their own writings.**
- **But competent writers do a lot of editing of their own work, correcting plenty of bugs.**
- **So do programmers:**
  - **Boris Beizer estimated that programmers make 150 bugs per 100 lines of code. We deliver about 0.5 to 5 bugs per 100 lines, so we (including our compilers, LINT, and other tools) apparently find / fix between 96.7% and 99.7% of our bugs.**
    - » Capers Jones provided some compatible data.
    - » I found confirming results in an informal (unpublished) 2-week self-study by about 10 programmers at Telenova in 1986.
- ***So what is it that testers are really looking for?***

# How do we find the bugs that are left?

We aren't just looking for bugs.

We're looking for the bugs that hide in programmers' blind spots.

# Programmers' Blind Spots?

- Testers work from collections of anecdotes about mistakes programmers make.
  - Some collections are published, like Whittaker's *How to Break Software*.
  - Some studies of programmers' errors have been done.
    - They'd be more useful if subjects were production programmers rather than undergraduates.
    - Even so, the questions of what mistakes do humans make, why do they make them, and why don't they find and correct them, are interesting psychological questions--of great benefit to testers.
  - Some statistical work on faults in real software has been done, especially Orthogonal Defect Classification.
    - But ODC pushes things into very broad categories. It tells us little about the mental causes or patterns. As a result, it provides testers with almost no guidance. (It may provide managerial guidance--but not what-faults-to-look-for or how-to-test-it guidance.)

# Let's Consider One Root Cause

- "It's a feature"
  - The programmer (include here the designer who wrote the spec) learned UI design somewhere, somewhen (maybe from someone who likes UNIX)
  - He intentionally wrote code that creates an unintended inconsistency:
    - E.G. the Gutenberg F-L, F-W, F-S; D-L, D-W, D-S.
  - Report the problem.
    - Response = "Don't do that."
- What's the solution?

# Scenario testing

- The ideal scenario has several characteristics:
  - The test is *based on a story* about how the program is used, including information about the motivations of the people involved.
  - The story is *motivating*. A stakeholder with influence would push to fix a program that failed this test.
  - The story is *credible*. It not only *could* happen in the real world; stakeholders would believe that something like it probably *will* happen.
  - The story involves a *complex use* of the program *or a complex environment or a complex set of data.*
  - The test results are *easy to evaluate.* This is valuable for all tests, but is especially important for scenarios because they are complex.

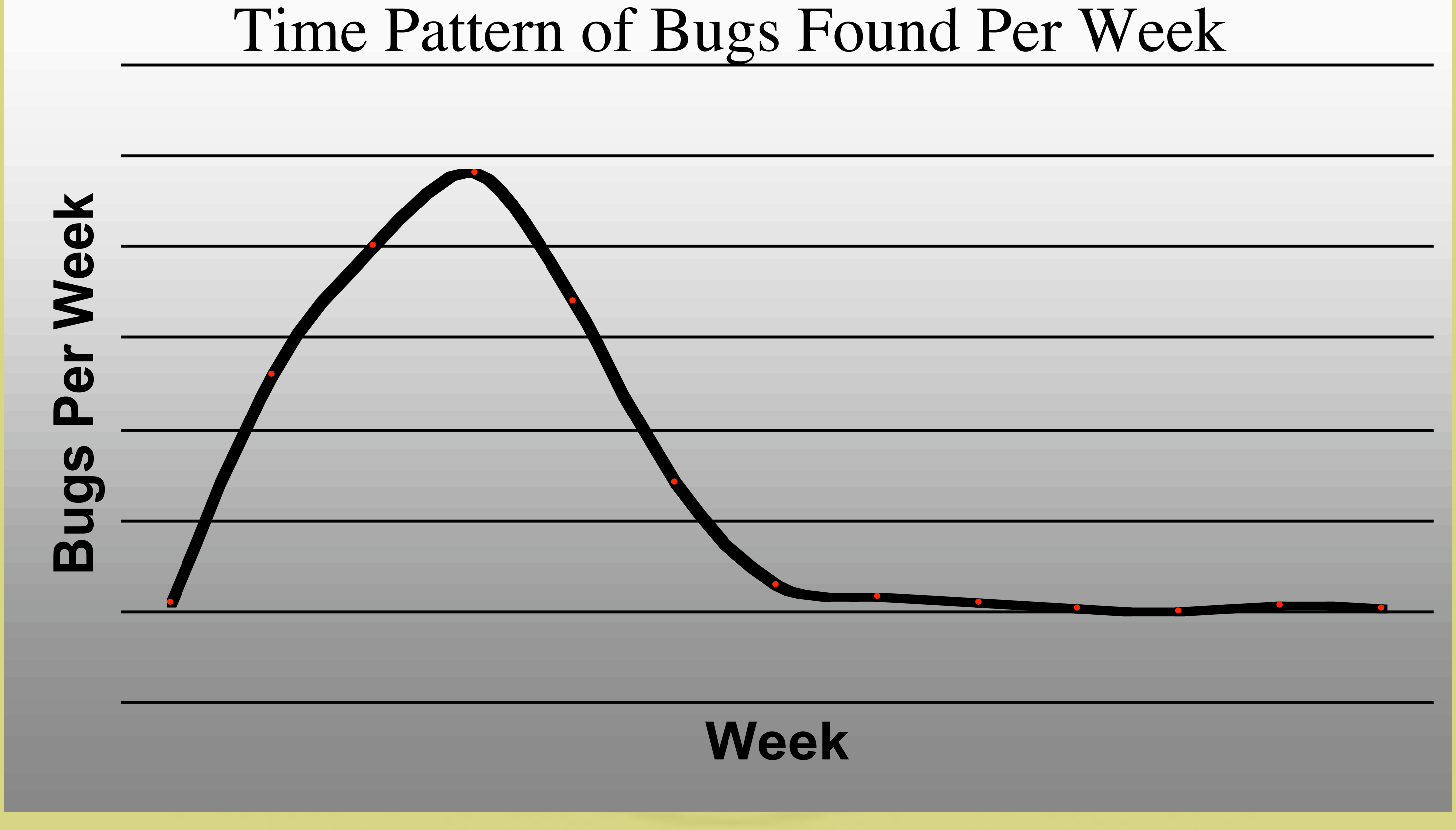
# Sixteen ways to create good scenarios

1. Write life histories for objects in the system. How was the object created, what happens to it, how is it used or modified, what does it interact with, when is it destroyed or discarded?
2. List possible users, analyze their interests and objectives.
3. Consider disfavored users: how do they want to abuse your system?
4. List system events. How does the system handle them?
5. List special events. What accommodations does the system make for these?
6. List benefits and create end-to-end tasks to check them.
7. Look at specific transactions that people try to complete, such as opening a bank account or sending a message. List all the steps, data items, outputs, displays, etc.?
8. What forms do the users work with? Work with them (read, write, modify, etc.)
9. Interview users about famous challenges and failures of the old system.
10. Work alongside users to see how they work and what they do.
11. Read about what systems like this are supposed to do. Play with competing systems.
12. Study complaints about the predecessor to this system or its competitors.
13. Create a mock business. Treat it as real and process its data.
14. Try converting real-life data from a competing or predecessor application.
15. Look at the output that competing applications can create. How would you create these reports / objects / whatever in your application?
16. Look for sequences: People (or the system) typically do task X in an order. What are the most common orders (sequences) of subtasks in achieving X?

# Scenarios

- Designing scenario tests is much like doing a requirements analysis, but is not requirements analysis. They rely on similar information but use it differently.
  - The requirements analyst tries to foster agreement about the system to be built. The tester exploits disagreements to predict problems with the system.
  - The tester doesn't have to reach conclusions or make recommendations about how the product should work. Her task is to expose credible concerns to the stakeholders.
  - The tester doesn't have to make the product design tradeoffs. She exposes the consequences of those tradeoffs, especially unanticipated or more serious consequences than expected.
  - The tester doesn't have to respect prior agreements. (Caution: testers who belabor the wrong issues lose credibility.)
  - The scenario tester's work need not be exhaustive, just useful.

# Let's Play with Another Myth:
# We Can(?) Measure Project Status Using Bug Counts



**This curve is used to predict ship dates and to signal when a milestone has or has not been met. The predictions are based on fitting bug counts to a Weibull curve. What's the rationale?**

# The Weibull Model--Absurd Assumptions

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

Based on Lyu, 1995; From Simmons, 2000

# Side effects of bug curves

- The problem with metrics that are only loosely tied to the attribute that they allegedly measure is that it is too easy to change the measured number without improving the underlying attribute.
- As a result, we get measurement distortion or dysfunction.

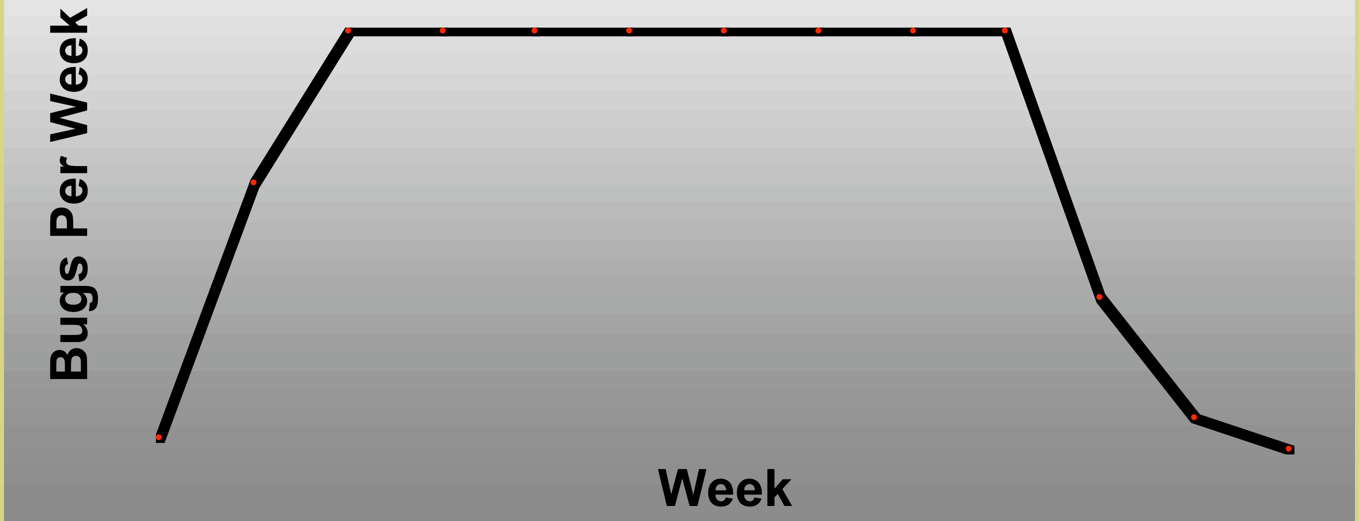
---

- Earlier in testing: (Pressure is to increase bug counts)
  - Run tests of features known to be broken or incomplete.
  - Run multiple related tests to find multiple related bugs.
  - Look for easy bugs in high quantities rather than hard bugs.
  - Less emphasis on infrastructure, automation architecture, tools and more emphasis of bug finding. (Short term payoff but long term inefficiency.)
    - » For more, see Hoffman's Dark Side of Metrics, and Austin's Measuring & Managing Performance in Organizations

# Side effects of bug curves

- Later in testing: Pressure is to decrease *new bug* rate
  - Run lots of already-run regression tests
  - Don’t look as hard for new bugs.
  - Shift focus to appraisal, status reporting.
  - Classify unrelated bugs as duplicates
  - Class related bugs as duplicates (and closed), hiding key data about the symptoms / causes of the problem.
  - Postpone bug reporting until after the measurement checkpoint (milestone). (Some bugs are lost.)
  - Report bugs informally, keep them out of the tracking system
  - Testers get sent to the movies before measurement checkpoints.
  - Programmers ignore bugs they find until testers report them.
  - Bugs are taken personally.
  - More bugs are rejected.

# ***Shouldn't We Strive For <u>This</u> ?***



Rather than accepting the declining find rate, the *Generic Test Strategy* (end of the slides) says, when the find rate drops, change your focus or change your technique. That way, you pull the find rate back up.

As a measure of progress, quality, or effectiveness, bug counts are oversimplified, overused, and untrustworthy.

# A Traditional Vision of Software Testing

- We start with an agreed product definition.
  - The "definition" includes the set of requirements, design, data, interoperability (and maybe some other) specifications
  - We may have helped shape the definition during reviews of requirements and specifications.
- We will base system testing on this definition. Therefore:
  - Analysis of assertions included in the documents is an important skill, and an interesting area for automation.
  - Traceability of tests back to the definition is important. Tools that support traceability have significant value.
  - Authoritative specifications are vital. If they are unavailable, it is the task of the test group to engage in "political" activities to motivate creation of suitable documents.

# A Traditional Vision of Software Testing

- The goal of system testing is verification of the product against authoritative specifications.
  - Our test population is a portion of the infinite population of potential tests. We select a minimally sufficient set of tests via analysis of the specifications.
  - Coverage is achieved by having tests for every specification item. An alternative type of coverage includes all length-N subpaths, where paths are from operational mode (observable state) to mode.
  - Testing progress is measurable in terms of % of planned tests coded and passed.
  - System testing is labor intensive. To the maximum extent possible, we automate our tests, enabling efficient reuse.
  - If test execution is expensive for a given project, we need selection algorithms (and tools) that help us choose the regression test subset most relevant for testing the program in the context of the latest change.

# A Traditional Vision of Software Testing

- In addition to system testing, we have programmer testing (aka "unit testing", though it is not restricted to true units)
- The traditional goal of programmer testing is detection of misimplementations, such as errors in:
  - syntax
  - branching (to missing or wrong place)
  - interrupt handling
  - coding of a possibly complex logical relationship
  - interpretation of incoming data
  - error anticipation and control (e.g. missing or wrong boundary value)
  - packaging of outgoing data
- hand-coded unit testing and more heavily automated mutation tests and protocol tests typify this work.
- Coverage is achieved structurally (test all lines, branches, or logical conditions) or by testing all 1st through Nth order data flows.

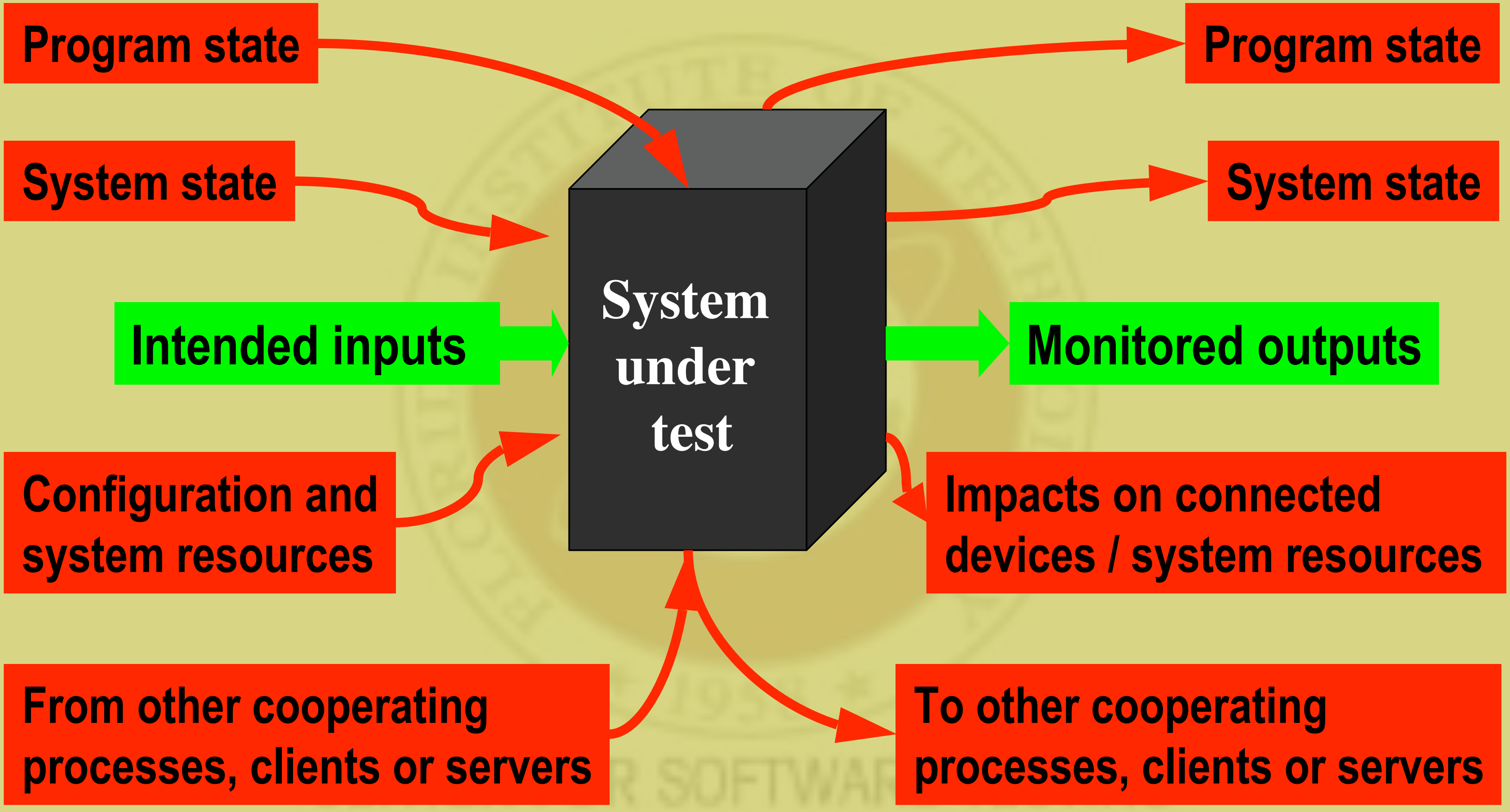
# I Live in a Slightly Different World From This

- The reference definition is:
    - not complete
    - not authoritative
    - not correct
    - not frozen
- The reference has to please multiple stakeholders
    - collectively they want more than time/budget possible
    - they have conflicting interests and needs
    - they turn over and are replaced by new people with different agendas, interests, and needs, and who don't consider themselves as parties to earlier compromises
- The reference assumptions and decisions are challenged by
    - implementation difficulties
    - usability problems ("it seemed to make sense on paper")
    - emergent properties that create unexpected almost-consistencies and other confusions, user frustration, or new almost-there usage opportunities

# I Live in a Slightly Different World From This

- Because the reference definition (if it exists) is non-authoritative, testing is not fully derived from the reference definition and test design might not rely on it at all.
- In addition, the same specified-item can be tested in many ways
  - with different data (data variation we intend)
  - in context of different paths (prior tasks may have changed data, resource availability, timing or other things we had not intended to change)
- The program can fail in many different ways, and our tests cannot practically address all possibilities

# The Program Can Fail in Many Ways



Program state

System state

Intended inputs

Configuration and system resources

From other cooperating processes, clients or servers

System under test

Program state

System state

Monitored outputs

Impacts on connected devices / system resources

To other cooperating processes, clients or servers

Based on notes from Doug Hoffman

# The Program Can Fail in Many Ways

- The phenomenon of inattentional blindness
  - humans (often) don't see what they don't pay attention to
  - programs (always) don't see what they haven't been told to pay attention to
- This is often the cause of irreproducible failures. We paid attention to the wrong conditions.
  - But we can't pay attention to all the conditions
- The 1100 embedded diagnostics
  - Even if we coded checks for each of these, the side effects (data, resources, and timing) would provide us a new context for the Heisenberg principle

## Our Tests Cannot Practically Address All of the Possibilities

# And *Even If* We Demonstrate a Failure That Doesn't Mean Anyone Will Fix It

- The comments on Sunday,
    - "Why didn't they find that?"

  were amusing because
    - "They" probably did find it!
- What distinguishes better and worse software publishers is not whether they try to fix all their bugs but rather:
  - the quality of the cost / benefit analysis they use to decide what to (not) fix, and
  - the extent to which they accept responsibility for their failures, rather than externalizing their failure costs
- Of course, the quality of the *bug description* will influence the should-we-fix-it analysis

# Testers Operate Under Uncertainty

- Testers have to learn, for each new product:
  - What are the goals and quality criteria for the project
  - What is in the product
  - How it could fail
  - What the consequences of potential failures could be
  - Who might care about which consequence of what failure
  - How to trigger a fault that generates the failure we're seeking
  - How to recognize failure
  - How to decide what result variables to pay attention to
  - How to decide what *other* result variables to pay attention to in the event of intermittent failure
  - How to troubleshoot and simplify a failure, so as to better
    - (a) motivate a stakeholder who might advocate for a fix
    - (b) enable a fixer to identify and stomp the bug more quickly
  - How to expose, and who to expose to, undelivered benefits, unsatisfied implications, traps, and missed opportunities.

# Bach's Test Strategy Model

**Quality Factors**

- Capability
- Reliability
- Security
- Safety
- Usability
  - Error induction
  - Error recovery
- Performance
- Concurrency
- Scalability
- Maintainability
- Installability
- Compatibility
- etc

**Product Elements**

- The physical product (code, interfaces, other files, etc.)
- Functions
- Data (input, output, preset, persistent)
- Platform (external hardware and software_
- Temporal relationships
- Operations: how it is used, who uses it, usage profiles
- etc

**Project Environment**

- Customers of the product
- Customers of the testing effort
- Project stakeholders
- Test-useful information
- Equipment & tools
- Test artifacts from prior projects
- Required deliverables
- Logistics
- Budget
- etc

**Potential vulnerabilities**

- Failure modes. Example--time-out in transferring data to a server causes failed transaction.
- Common faults: Example--boundary miscoded (see Whittaker's attacks in *How to Break Software)*
- Project risks: Example: Programmer of a given component has a substance abuse problem at work

# Making Time for Usability Testing / Fixing

- Complete testing requires infinite time, so we prioritize.
- We prioritize according to the interest levels we expect from our stakeholders.
  - We don't want to spend much time looking for bugs that none of them care about, and none of them will fix.
- One reason stakeholders are resistant to usability testing--especially user error reduction--is these issues can be externalized.
  - It's YOUR fault. YOU made the mistake. YOU pay for the consequences.
- In addition, new users are gradually trained into learned helplessness, by the press and by appalling technical support.
- To advocate for change, at the tester level, is to find compelling examples of failures caused by the class of problem. Normally, we cast these as scenarios. Compelling scenarios get stuff fixed, and justify time spent looking for more.
- From a systems-level point of view, I think that if you want these problems addressed systematically, you have to create accountability. That's a legal issue, though, not a testing issue, so it's out of scope for today's talk.

# Software Testing as a Social Science

Cem Kaner

Florida
Institute of
Technology

Siena, Italy

IFIP
Working
Group 10.4

Software
Dependability

July, 2004

CENTER FOR SOFTWARE TESTING
EDUCATION AND RESEARCH

# What’s a test case?

- Focus on procedure?
  - “A set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement.” (IEEE)
- Focus on the test idea?
  - “A test idea is a brief statement of something that should be tested. For example, if you're testing a square root function, one idea for a test would be ‘test a number less than zero’. The idea is to check if the code handles an error case.” (Marick)

**These are NOT part of the main talk, but they tie into it and some readers might find them useful.**

# Test cases

- In my view,

**A test case is a question you ask of the program.**

- The point of running the test is to gain information, for example whether the program will pass or fail the test.
- Implications of this approach:
  - The test must be CAPABLE of revealing valuable information
  - The SCOPE of a test changes over time, because the information value of tests changes as the program matures
  - The METRICS that count test cases are essentially meaningless because test cases merge or are abandoned as their information value diminishes.

# A Generic Strategy for Testing Fallible Advice for Newcomers

- **When you start testing a new program, start with sympathetic tests (tests intended to help you understand what the program is designed to do.) Test every function, using mainstream values rather than boundaries. Keep it simple and broad.**
- **When the program can pass the elementary tests, use boundary tests on every input / output / intermediate calculation variable you can find. Then test 2-variable or 3-variable combinations of boundaries of related and allegedly unrelated variables.**

***Books on testing often present this as the only test design technique, or the only design technique they spell out. But look at how narrow its scope is--one variable, or a few (2, 3 or 4). Contrast this with what we know about the multidimensionality of the failures we reviewed on Sunday.***

***This technique marks a start, not an end.***

# A Generic Strategy for Testing
# Fallible Advice for Newcomers

- When the program can pass boundary tests, start serious testing. Your next steps depend on your context:
  - If you have an influential specification, start with specification testing.
  - If you have good lists of failure modes (from customer complaint records, other products, or a failure mode analysis), start with risk-based testing, designing tests to check the failure modes.
  - If you have an expert user at hand, good descriptions of the ways people use the product, or UML specifications, start with scenario testing.
  - If you want to assess the competence of the programming team and potentially find a large number of bugs in a hurry, start with Whittaker and Jorgenson's attacks (see Whittaker's how to break software).
- There are more contingent recommendations, but these give the flavor.

# A Generic Strategy for Testing
# Fallible Advice for Newcomers

- As you test with a given style, you will find problems at a satisfactory rate or not.
  - If not, switch the focus of your tests or switch to a new technique.
- As you test more, test with more complex combinations. These might simulate experienced users' tasks or they might involve complex sequences (probably automated) to check for  tough faults like wild pointers, memory leaks, or races.
- The more stable the program gets, the harder you can push it. Keep pushing until you run out of good test ideas. Bring in new people with new skills. At the point that your best ideas are ineffective, stop testing and ship the product.

# A Generic Strategy for Testing Fallible Advice for Newcomers

- Under this strategy, last week's tests are boring. We must set up some regression testing (e.g. smoke testing) but if your company practices source control and if the code is not fundamentally unmaintainable, then the less repetition of previous tests that we do, the better.
- This doesn't mean that we don't go back to old areas. It means that we spiral through them. The bottom of the spiral are the single-function tests. Cover every feature. Spiral through all the features again with boundary tests. And again with each new technique. Higher spirals combine many features / many areas / many variables in the same test case.
- If changes destabilize existing code, run simpler tests of that code until it restabilizes. As it regains credibility, go back to tests that are harsh and complex enough to provide information (reduce uncertainty).

# Test design

- Differentiating between
  - User testing
    - We run tests with representatives of the user community.
      - The term "user testing" tells us nothing about the scope or intent or coverage of the tests, the way the tests will be run, the bugs you're looking for (the risks you're mitigating), or how you will recognize a bug.
  - User interface testing
    - We test the user interface elements, such as the various controls. "Complete" testing would cover all the elements.
      - We know what we're testing, but not how, what bugs we're looking for, how to recognize a bug, or who will do the testing.
  - Usability testing
    - We test the product for anything that makes it less usable. This includes traps, aspects of the user interface that lead a person to error. It also includes poor recovery from user error.

**These notes are based on our discussion in Lessons Learned in Software Testing. The relevance to this talk is my comment that different test objectives cause you to use a different mix of techniques. OK, so what are techniques? All three of the examples on this page have been called techniques. What do they tell us?**

# Test Design

**The act of testing involves doing something that resolves:**

- Tester*: who* **does the testing.**
- Coverage*: what* **gets tested.**
- Potential problems*: why* **you're testing (what risk you're testing for).**
- Activities*: how* **you test.**
- Evaluation*: how to tell whether the test passed or failed.*

*All testing involves all five dimensions.*

# Test Techniques

1. A test technique is a plane in the 5-dimensional space. It specifies values on one or more dimensions
    - (e.g. *user testing* specifies who tests, but nothing on the other dimensions)
    - (*e.g.* input-boundary-value testing specifies what you test (input variables) and a risk (for some values of the variable, the program will halt or branch to the wrong place)

- A test technique is a heuristic.
    - There are a couple hundred (at least) named test techniques.
    - They are useful at different times, for different things.
    - Some are taught more widely than others, some are Officially Recognized (e.g. SWEBOK), but they are all just tools that are sometimes helpful.

# Heuristics

Billy V. Koen, Definition of the Engineering Method, ASEE, 1985

- A heuristic is anything that provides a plausible aid or direction in the solution of a problem but is in the final analysis unjustified, incapable of justification, and fallible. It is used to guide, to discover, and to reveal.
- Heuristics do not guarantee a solution
- Two heuristics may contradict or give different answers to the same question and still be useful
- Heuristics permit the solving of unsolvable problems or reduce the search time to a satisfactory solution
- The heuristic depends on the immediate context instead of absolute truth as a standard of validity.

Koen offers an interesting definition of engineering

  – *The engineering method is the use of heuristics to cause the best change in a poorly understood situation within the available resources* (p. 70).