

Thinking about the Software Testing Curriculum

Cem Kaner, J.D., Ph.D.

Presentation at
Florida International University
Miami, March 2009

Copyright (c) Cem Kaner 2008-2009

This work is licensed under the Creative Commons Attribution-ShareAlike License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-sa/2.0/> or send a letter to Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

These notes are partially based on research that was supported by NSF Grants EIA-0113539 ITR/SY+PE: "Improving the Education of Software Testers" and CCLI-0717613 "Adaptation & Implementation of an Activity-Based Online or Hybrid Course in Software Testing." 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.

System Testing	Programmer Testing
<ul style="list-style-type: none"> • Does the program meet the needs of the stakeholders? • Evidence is taken from every source that provides information about the needs and preferences of the stakeholders (requirements documents, tech support data, competing products, interviews of stakeholders, etc.) • Tests are typically behavioral. For practical reasons they are usually black box (a subspecies of behavioral). Also, for psychological reasons--focus the tester on the stakeholder. • Tools are diverse. GUI regression tests are common but wasteful. More useful to think in terms of computer-assisted testing. • High volume tools are in infancy, but vital 	<ul style="list-style-type: none"> • Does the program do what I intended? • Evidence is taken from the programmer's intent, which might be reflected in design documents, unit tests, comments, or personal memory • Tests are almost always glass box, though in practice, they are often runs of the working program while reviewing a listing or running a debugger • Tools: Unit test frameworks (e.g. JUNIT), code coverage, complexity metrics, version control, source code analyzers, state models

System Testing	Programmer Testing
<ul style="list-style-type: none"> • About 20% to 60% of the new product development effort (in terms of staff size) • This is NOT primarily about programming. To a very large degree, this is applied social science, plus specific subject matter expertise. (Of course, programming skills help in many ways: realistic theory of error; communication quality; tool use) 	<ul style="list-style-type: none"> • All programmers do programmer testing to some degree. Even weak programmers find the vast majority of their own bugs (public vs private bugs) • This IS programming. This helps the programmer understand her implementation (or the implementation by a colleague). • The tools are easy. What to DO with the tools is hard: <ul style="list-style-type: none"> • Problem decomposition • Discrete math (including graphs) • Boolean logic (complex combinations)

Curricular objectives

- Development of the IEEE/ACM curriculum guide for SE used the dumpster algorithm.
- Resulting recommendations are for a course that is broad and shallow.
 - Learn lots of definitions
 - Get "familiar" with lots of concepts
 - Get skilled at almost nothing
 - I don't think these types of courses have scholarly or academic merit.
- Recent texts meet the curriculum guide requirements, adding detail that corresponds to the authors' biases about what is important in the field. (Lots of applied mathematics, or lots of standards-compliance, not much stakeholder value.)
- I think everyone is served better by a tighter focus

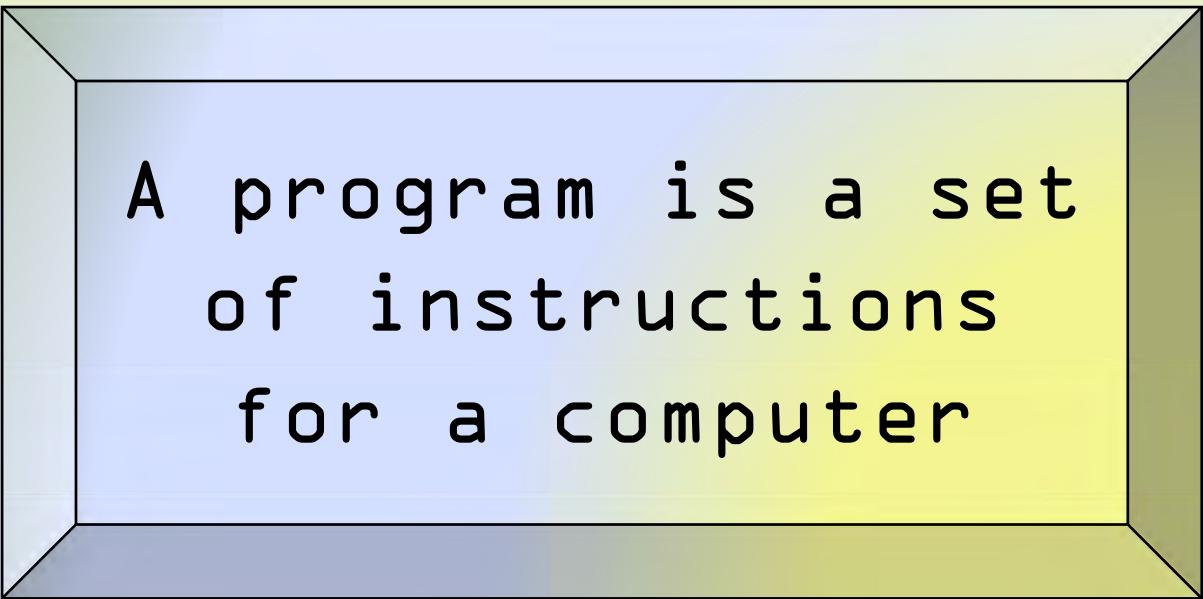
System Testing Course	Programmer Testing Course	Programming Course that includes testing
<ul style="list-style-type: none"> • quality as stakeholder value • test-applied requirements analysis • heuristics for telling "failure" from "pass" • reporting bugs effectively • mining data from complex sets of documents • failure modes and consequences from an external view • measuring and reporting progress • techniques (domain (risk-aware stratified sampling), risk-based, spec-based, scenarios, etc.), qualitative analysis • using tools cost-effectively 	<ul style="list-style-type: none"> • basic tools: professional, integrated programming environment, xUnit, coverage monitors (many types of coverage), style checkers, version control, source code analyzers • applying the tools to significant programming tasks (fresh code and maintenance) • critical problems include inability or unwillingness to decompose problems, resistance, and limited imagination about what to test (what tests are interesting) • few recent books on programmer testing address test design well 	<ul style="list-style-type: none"> • easy to introduce the tools into the labs • what are you going to have students DO with them?

*A system
tester's view of
the world*

What's a Computer Program?

The last couple of years, I taught intro programming.

Texts define a “computer program” like this:



A program is a set
of instructions
for a computer

Computer Program

A set of instructions for a computer?

What about what the program is for?

Computer Program

A set of instructions for a computer?

What about what the program is for?

We could define a house

- as a set of construction materials
- assembled according to house-design patterns.

Computer Program

A set of instructions for a computer?

What about what the program is for?

We could define a house

- as a set of construction materials
- assembled according to house-design patterns.

*But I'd rather define it as something
built for people to live in.*

Something built for people to live in...

The focus is on

- Stakeholders
 - (for people)
- Intent
 - (to live in)

Stakeholder:

A person who is affected by:

- the success or failure of a project
- or the actions or inactions of a product
- or the effects of a service.

Set of instructions for a computer...

Where are the

- Intent?
- Stakeholders?

*Ignore these and lay
groundwork for the
classic problems of
software engineering ...*

*in the first week of
the first programming
class.*

A different definition

A computer program is

- a communication
- among several humans and computers
- who are distributed over space and time,
- that contains instructions that can be executed by a computer.

The point of the program is to provide value to the stakeholders.

Social Science?

Social sciences study humans, especially humans in society.

- What will the impact of X be on people?
- Work with qualitative & quantitative research methods.
- High tolerance for ambiguity, partial answers, situationally specific results.
- Ethics / values issues are relevant.
- Diversity of values / interpretations is normal.
- Observer bias is an accepted fact of life and is managed explicitly in well-designed research.

What are we really testing for?

Quality is value to some person

-- Jerry Weinberg

Under this view:

- Quality is inherently subjective
 - Different stakeholders will perceive the same product as having different levels of quality

Testers look
for different
things ...
... for different
stakeholders

Software error

An attribute of a software product

- that reduces its value to a favored stakeholder
- or increases its value to a disfavored stakeholder
- without a sufficiently large countervailing benefit.

An error:

- May or may not be a coding error
- May or may not be a functional error

Any threat to
the value of
the product to
any stakeholder
who matters.

James Bach

What are we really testing for?

Quality is value to some person

-- Jerry Weinberg

But is every limitation on value an error?

Is a car defective if it can't withstand a 40 mph crash into a brick wall?

www.tanksandjeeps.com T55 tanks for sale page - Mozilla Firefox 3 Beta 4

File Edit View History Bookmarks Tools Help

http://www.tanksforsale.co.uk/T55 Tanks for sale_page.htm

buy a used tank ABP

www.tanksforsale.co.uk

Tanks, Jeeps, Trucks for sale, hire & export.

Tel: England 0044 (0)1772 601214

Our Company Site Map For Sale: Tanks, Jeeps, Trucks For Hire: Tanks, Jeeps, Trucks Import / Export, Shipping / Transport Evaluation + Brokerage Service Prices & FAQs Email Us

Flags: Czech Republic, Poland, Russia, Hungary

Icons: Person, Staff, Vehicles, Party, Tools, Help, Mail

T55 AM2 Tanks for sale

With the T55 you are buying what is probably the most reliable and simple to maintain Main Battle Tank for sale in the UK. The vehicles we are importing are (unlike many Ex Gulf War veterans elsewhere for sale in the UK on Milweb) in good condition with very low running hours and KM. This tank is recommended for collectors, and off road enthusiasts alike. The Russian T55 AM2 Tanks for sale are the latest uparmoured version of the T55.

T55 For Sale - import licence is in progress December 2006. ETA UK Jan 07.

T55 Additional armour on turret front

T55 Turret side with smoke grenade dischargers

T55 Engine Compartment

Not every limitation on value is a bug:

Effective bug reporting requires evaluation of the product's context (market, users, environment, etc.)

Software testing

- is an empirical
- technical
- investigation
- conducted to provide stakeholders
- with information
- about the quality
- of the product or service under test

We design and run tests in order to gain useful information about the product's quality

Testing is always a search for information

- 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 product support costs
- 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 require different testing tools and strategies and will yield different tests, different test documentation and different test results.

Test techniques

A test technique is essentially a recipe, or a model, that guides us in creating specific tests. Examples of common test techniques:

- Function testing
- Specification-based testing
- Domain testing
- Risk-based testing
- Scenario testing
- Regression testing
- Stress testing
- User testing
- All-pairs combination testing
- Data flow testing
- Build verification testing
- State-model based testing
- High volume automated testing
- Printer compatibility testing
- Testing to maximize statement and branch coverage

We pick the technique that provides the best set of attributes, given the information objective and the context.

Examples of test techniques

- **Scenario testing**
 - Tests are complex stories that capture how the program will be used in real-life situations.
- **Specification-based testing**
 - Check every claim made in the reference document (such as, a contract specification). Test to the extent that you have proved the claim true or false.
- **Risk-based testing**
 - A program is a collection of opportunities for things to go wrong. For each way that you can imagine the program failing, design tests to determine whether the program actually will fail in that way.

Techniques differ in how to define a good test

Power. When a problem exists, the test will reveal it

Valid. When the test reveals a problem, it is a genuine problem

Value. Reveals things your clients want to know about the product or project

Credible. Client will believe that people will do the things done in this test

Representative of events most likely to be encountered by the user

Non-redundant. This test represents a larger group that address the same risk

Motivating. Your client will want to fix the problem exposed by this test

Maintainable. Easy to revise in the face of product changes

Repeatable. Easy and inexpensive to reuse the test.

Performable. Can do the test as designed

Refutability: Designed to challenge basic or critical assumptions (e.g. your theory of the user's goals is all wrong)

Coverage. Part of a collection of tests that together address a class of issues

Easy to evaluate.

Supports troubleshooting. Provides useful information for the debugging programmer

Appropriately complex. As a program gets more stable, use more complex tests

Accountable. You can explain, justify, and prove you ran it

Cost. Includes time and effort, as well as direct costs

Opportunity Cost. Developing and performing this test prevents you from doing other work

Techniques differ in how to define a good test

- **Scenario testing:**
 - complex stories that capture how the program will be used in real-life situations
 - Good scenarios focus on validity, complexity, credibility, motivational effect
 - The scenario designer might care less about power, maintainability, coverage, reusability
- **Risk-based testing:**
 - Imagine how the program could fail, and try to get it to fail that way
 - Good risk-based tests are powerful, valid, non-redundant, and aim at high-stakes issues (refutability)
 - The risk-based tester might not care as much about credibility, representativeness, performability—we can work on these after (if) a test exposes a bug

Examples of important context factors

- Who are the stakeholders with influence
- What are the goals and quality criteria for the project
- What skills and resources are available to the project
- What is in the product
- How it could fail
- Potential consequences of potential failures
- Who might care about which consequence of what failure
- How to trigger a fault that generates a failure we're seeking
- How to recognize failure
- How to decide what result variables to attend to
- How to decide what *other* result variables to attend to in the event of intermittent failure
- How to troubleshoot and simplify a failure, so as to better
 - motivate a stakeholder who might advocate for a fix
 - 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.

Software testing

- is an empirical
- technical
- investigation
- conducted to provide stakeholders
- with information
- about the quality
- of the product or service under test

There might be as many as 150 named techniques.

**Different techniques are useful
to different degrees
in different contexts**

A Toxic Myth about Testing: Testing = Verification

**NOTICE THE HUGE DIFFERENCE HERE
BETWEEN PROGRAMMER TESTING AND
SYSTEM TESTING**

IF you have contracted for delivery of software, and the contract contains *a complete and correct specification*,

THEN verification-oriented testing can answer the question,

Do we have to pay for this software?

Verification is insufficient for commercial software

Verification-oriented testing can answer the question:

Do we have to pay for this software?

But if...

- You're doing in-house development
- With evolving requirements (and therefore an incomplete and non-authoritative specification).

Verification only begins to address the critical question:

Will this software meet our needs?

Verification / Validation

In system testing,

the primary reason we do verification testing is to assist in:

- **validation:**

Will this software meet our needs?

- or **accreditation:**

Should I certify this software as adequate for our needs?

Does it really matter whether the program
meets its specification,
if it won't meet our needs?

System testing (validation)

Designing system tests is like doing a 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 system tester's work cannot be exhaustive, just useful.

It's kind of like CSI

The screenshot shows the official website for the TV show CSI. At the top, there's a navigation bar with links for 'EPISODES', 'HANDBOOK', 'VIDEO', 'PRODUCTION', and 'INTERACTIVE'. A dropdown menu says 'Choose a CBS Show' and has an 'Adblock' option. Below the navigation is a banner for 'CSI: CRIME SCENE INVESTIGATION THURSDAYS 9PM ET/PT'. The main content area has a sidebar titled 'HANDBOOK' with links to various tools and procedures. One section, 'AFIS', is highlighted and shown in a larger window. This window contains a thumbnail of a fingerprint and a detailed description of the AFIS system. At the bottom of the page, there are sections for 'CSI HANDBOOK', 'PREVIOUSLY ON CSI' (with a thumbnail for 'Room Service'), 'CSI VIDEO' (with a thumbnail for 'Behind the Scenes'), and 'CSI NEWSLETTER'.

MANY tools, procedures, sources of evidence.

- Tools and procedures don't define an investigation or its goals.
- There is too much evidence to test, tools are often expensive, so investigators must exercise judgment.
- The investigator must pick what to study, and how, in order to reveal the most needed information.

Some examples

I probably won't reach these in the talk (not enough time for them), but they are worth taking some time to consider if you are reading these slides.

- Bug reporting is one of the critical skills in software testing, but the communication skills needed for this are taught weakly in the CS curriculum
- IEEE standards and DoD approaches to testing favor heavily scripted tests. These ARE slightly better than worthless, but they are enormously expensive and grossly inadequate. They also provide weaker controls for junior testers than you might expect.
- Assessing whether a program passed or failed a test is usually a heuristic exercise. You can often tell whether some aspect of the program behaved as you expected, but that's a far cry from knowing whether the program is behaving correctly or misbehaving.
- Software engineering measurement deviates from theory of measurement in most other fields by underemphasizing construct validity (by not asking how we know whether the "measurement" actually measures what we are trying to measure). The result is predictable mischief, and the abandonment of metrics programs at most non-government-contractor companies.

Example I

Bug reporting is one of the critical skills in software testing, but the communication skills needed for this are taught weakly in the CS curriculum

Why aren't critical bugs fixed?

- Client experienced a wave of serious product recalls (defective firmware)
 - Why were these serious bugs not found in testing?
 - *They were found in testing and reported*
 - Why didn't the programmers fix them?
 - *They didn't understand what they were reading*
 - What was wrong with the bug reports?
 - **The testers focused on creating reproducible failures, rather than on the quality of their communication.**
 - Looking over 5 years of bug reports, I could predict deferrals better by clarity/style/attitude of report than from severity

What are we actually looking for / hoping to report?

- Blind spots
- Of significance to one or more stakeholders with influence

What are we actually looking for?

- **Blind spots**
 - Programmers find and fix most of their own bugs.
 - Testers find the bugs the programmers missed.
 - Therefore, the testing task = looking for the bugs that hide in programmers' blind spots.
 - To test effectively, our theories of error have to be theories about the mistakes people make and when / why they make them.
- We can and should take advantage of opportunities to routinize well-articulated theories of error (e.g. via mutation testing, fault injection) but these address only the theories of error (e.g. types of fault injected) that we explicitly consider.
 - As to the rest → blind spots.
- **By the way, what is coverage?**
 - % of tests executed . . .
 . . . out of the pool of tests that we can derive from the same theory of error

Coverage

- Common measures of coverage are weak guides for test designers:
 - 100% statement coverage will expose all syntax errors but achieves little device-configuration coverage
 - 100% functional-specification-statement coverage might achieve only 35% statement+branch coverage
 - 100% business scenario coverage might achieve little variable-boundary coverage
 - http://www.kaner.com/pdfs/negligence_and_testing_coverage.pdf
 - http://www.kaner.com/pdfs/measurement_segue.pdf
- Rather than mechanistically shooting for X% of Y type of coverage, practitioners explicitly or implicitly try for multidimensionally prioritized levels of coverage across different types of risks

**Where do we teach the core skills of prioritization?
More in the CS curriculum or the Business curriculum?**

What are we actually looking for / hoping to report?

- Blind spots
- **Of significance to one or more stakeholders with influence**
 - We investigate bugs in preparation for an effective report:
 - Discover
 - Isolate
 - Generalize
 - Maximize
 - Externalize
 - Tailor to audience
 - This lets us creates sales proposals (aka bug reports)
 - We are trying to motivate someone else to spend their resources to do something we want them to do.

Sales = software engineering?

- Persuasive communication comes up in many software engineering contexts.
- Famous examples
 - Challenger disaster
 - David Parnas' warnings on SDI (Star Wars)
 - Electronic voting equipment
- Routine example
 - Status reporting in the face of unreasonable demands (Death March)
- But if we **study the communication as a software engineering problem, how much traction does that give us?**
- Maybe we gain more insight from thinking about human-to-human communications (like, sales).

For more on my approach to bug advocacy

- See my bug advocacy videos (freely reusable) at:
 - <http://www.testingeducation.org/BBST/videos/BugAdvocacy2008A.wmv>
 - <http://www.testingeducation.org/BBST/videos/BugAdvocacy2008B.wmv>
 - <http://www.testingeducation.org/BBST/videos/BugAdvocacy2008C.wmv>
 - <http://www.testingeducation.org/BBST/videos/BugAdvocacy2008D.wmv>
 - <http://www.testingeducation.org/BBST/videos/BugAdvocacy2008E.wmv>
 - <http://www.testingeducation.org/BBST/videos/BugAdvocacy2008F.wmv>
- For model of additional instructional support, look at the Association for Software Testing courses (free to members)(\$50 membership) at <http://www.associationforsoftwaretesting.org/drupal/courses>
- (Soon, the supplementary course materials AST is developing for bug advocacy will move back to www.testingeducation.org/BBST, free availability, but the coached online instruction requires the course)

Example 2 & 3

- IEEE standards and DoD approaches to testing favor heavily scripted tests. These ARE slightly better than worthless, but they are enormously expensive and grossly inadequate. They also provide weaker controls for junior testers than you might expect.
- Assessing whether a program passed or failed a test is usually a heuristic exercise. You can often tell whether some aspect of the program behaved as you expected, but that's a far cry from knowing whether the program is behaving correctly or misbehaving.

Scripted testing

A script specifies

- the test operations
- the expected results
- the comparisons the human or machine should make

These comparison points are

- useful, but fallible and incomplete, criteria for deciding whether the program passed or failed the test

Scripts can control

- manual testing by humans
- automated test execution or comparison by machine

Key benefits of scripts

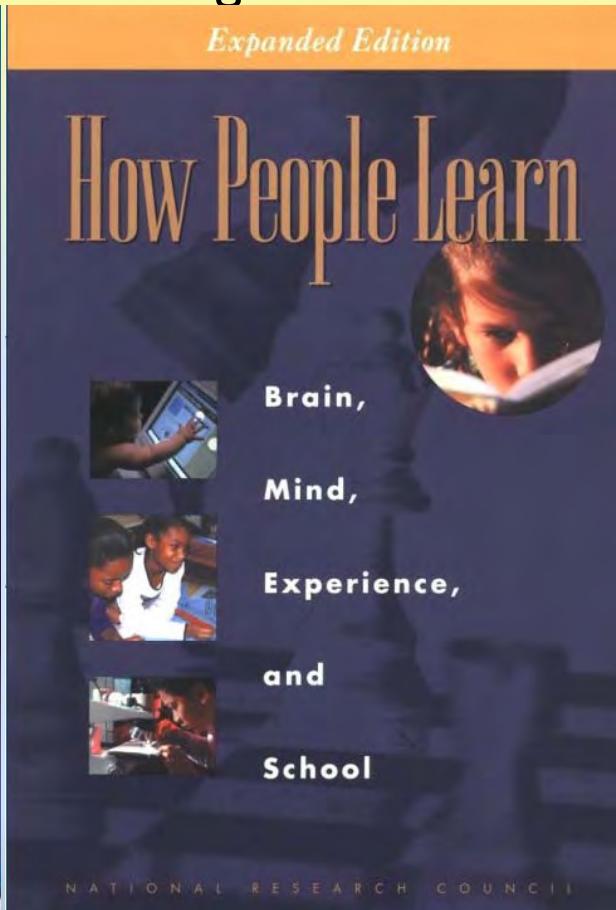
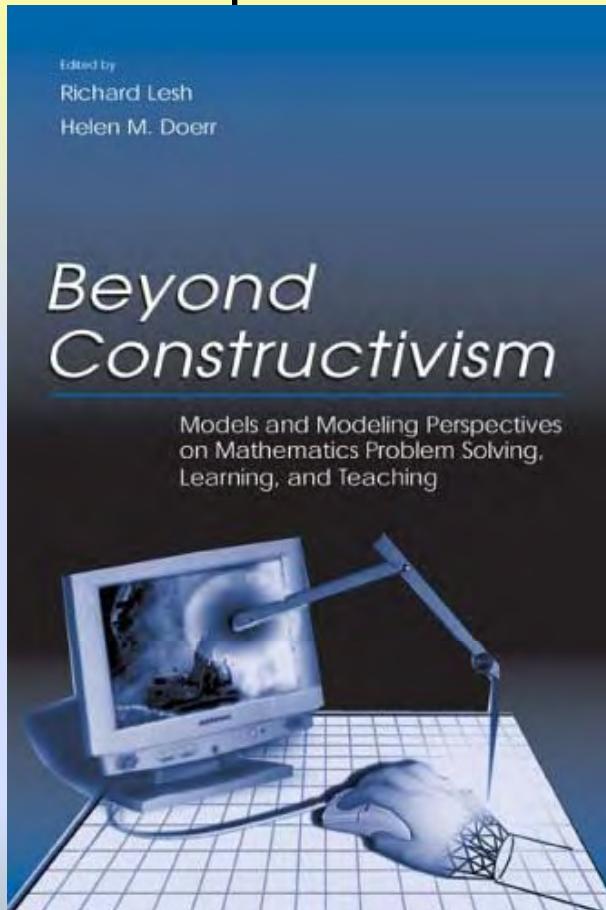
Scripts require a big investment. What do we get back?

The scripting process provides *opportunities* to achieve several key benefits:

- Careful thinking about the design of each test, optimizing it for its most important attributes (power, credibility, whatever)
- Review by other stakeholders
- Reusability
- Known comprehensiveness of the set of tests
- If we consider the set sufficiently comprehensive, we can calculate as a metric the percentage completed of these tests.

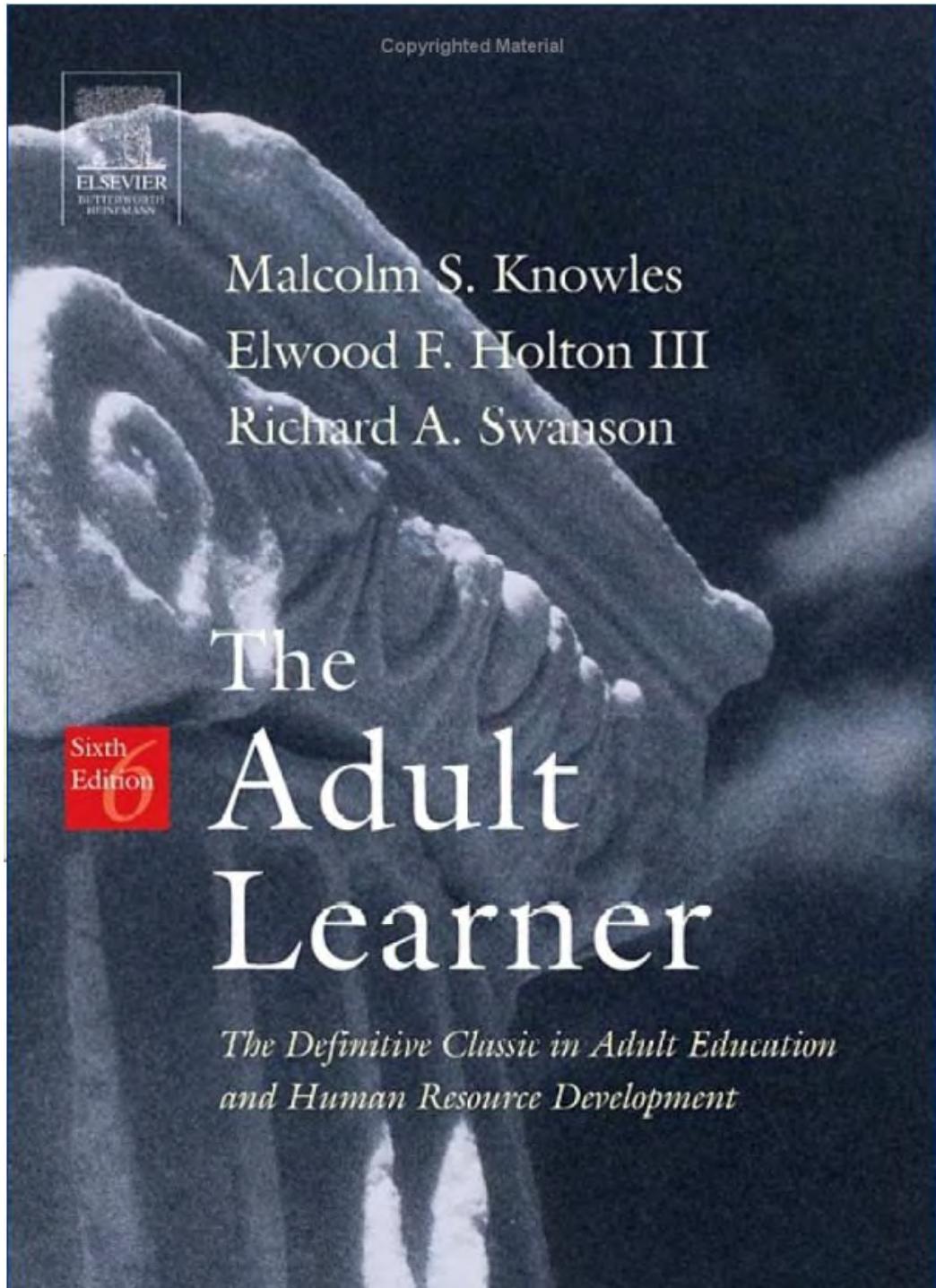
No, learning support is NOT a benefit of scripts

Justifiers of manual scripted testing often assert this is a good way to teach people about the software or about testing. These claims are incompatible with our knowledge of instructional design and learning.



In science / math education, the transfer problem is driving fundamental change in the classroom

Students learn (and transfer) better when they discover concepts, rather than by being told them



Scripts: Poor tools for adult learning

Pedagogy: study of teaching / learning of children

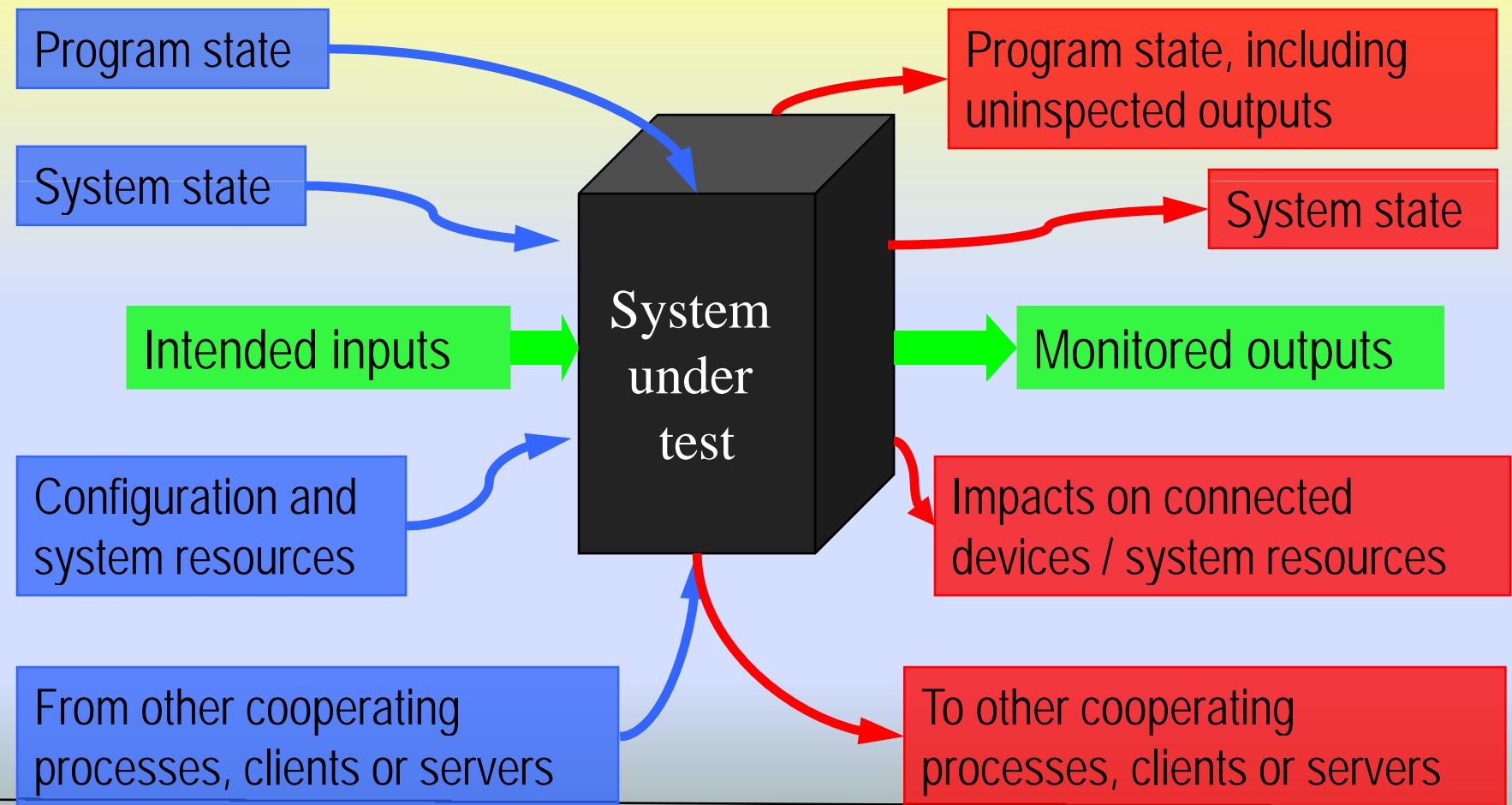
Andragogy: study of teaching / learning of adults

University undergrads are in a middle ground between the teacher-directed child and the fully-self-directed adult

Both groups, but especially adults, benefit from activity-based and discovery-based styles

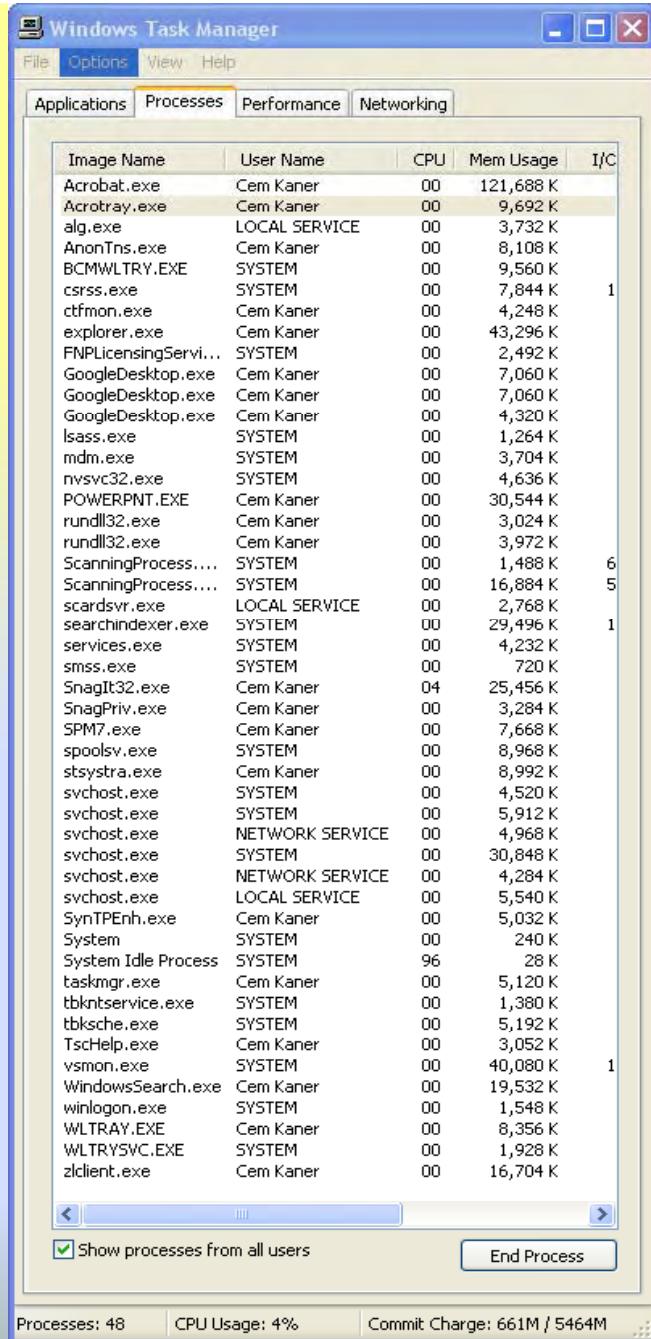
Problem with scripts: Programs fail in many ways

Based on notes from Doug Hoffman



Can you specify your test configuration?

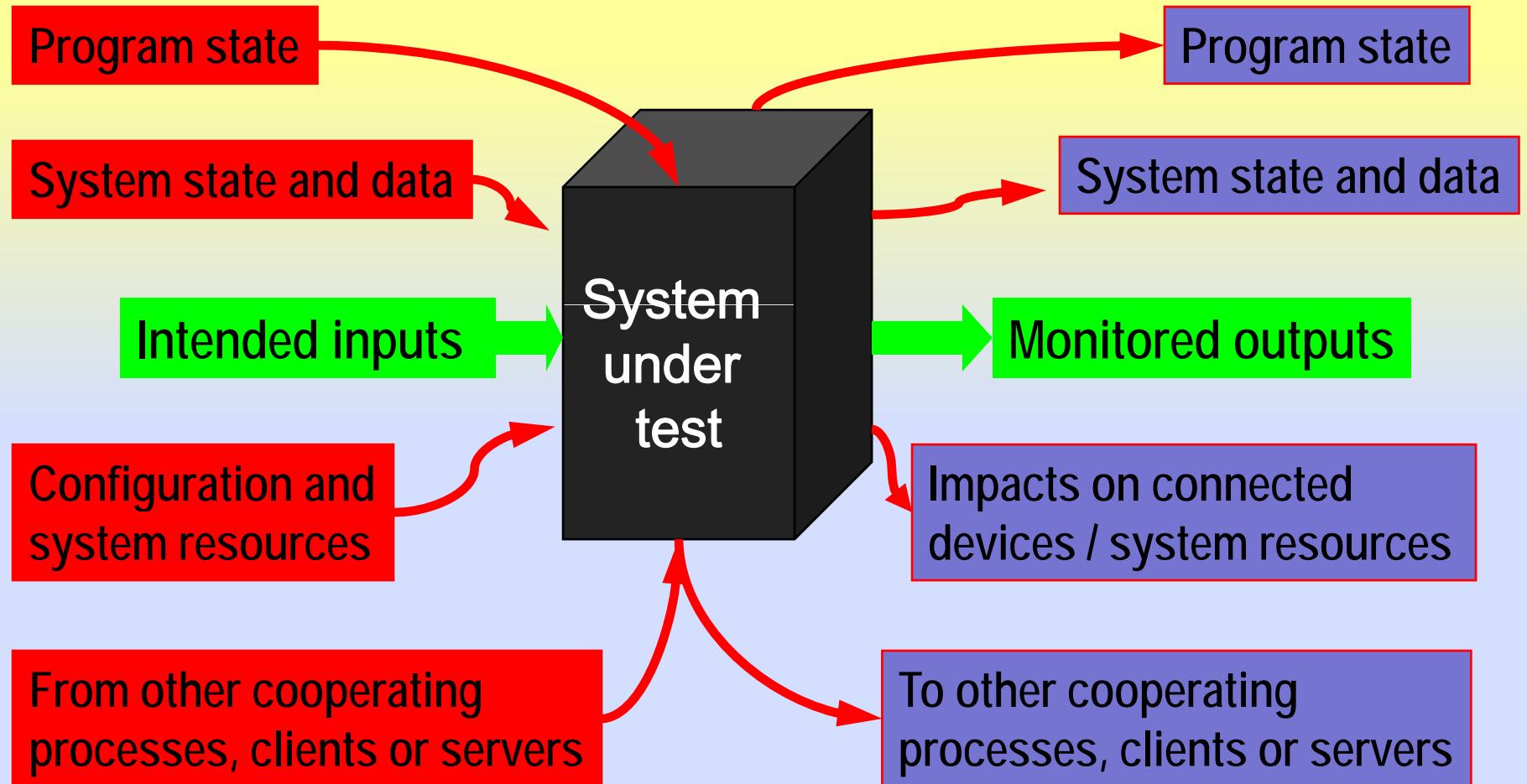
- Does your test documentation specify ALL of the processes running on your computer?
- Does it specify what version of each one?
- Do you even know how to tell
 - What version of each of these you are running?
 - When you (or your system) last updated each one?
 - Whether there is a later update?



The screenshot shows the Windows Task Manager window with the 'Processes' tab selected. The table lists various processes with their names, users, CPU usage, memory usage, and I/O count. The 'User Name' column shows mostly 'Cem Kaner' except for some SYSTEM and LOCAL SERVICE processes. The 'Mem Usage' column shows memory consumption in kilobytes (K). The 'I/C' column shows the number of input/output operations per second.

Image Name	User Name	CPU	Mem Usage	I/C
Acrobat.exe	Cem Kaner	00	121,688 K	1
Acrotray.exe	Cem Kaner	00	9,692 K	
alg.exe	LOCAL SERVICE	00	3,732 K	
AnonTns.exe	Cem Kaner	00	8,108 K	
BCMWLTRY.EXE	SYSTEM	00	9,560 K	
csrss.exe	SYSTEM	00	7,844 K	1
ctfmon.exe	Cem Kaner	00	4,248 K	
explorer.exe	Cem Kaner	00	43,296 K	
FNPLicensingServ...	SYSTEM	00	2,492 K	
GoogleDesktop.exe	Cem Kaner	00	7,060 K	
GoogleDesktop.exe	Cem Kaner	00	7,060 K	
GoogleDesktop.exe	Cem Kaner	00	4,320 K	
lsass.exe	SYSTEM	00	1,264 K	
mdm.exe	SYSTEM	00	3,704 K	
nvsvc32.exe	SYSTEM	00	4,636 K	
POWERPNT.EXE	Cem Kaner	00	30,544 K	6
rundll32.exe	Cem Kaner	00	3,024 K	
rundll32.exe	Cem Kaner	00	3,972 K	
ScanningProcess....	SYSTEM	00	1,488 K	5
ScanningProcess....	SYSTEM	00	16,684 K	
scardsvr.exe	LOCAL SERVICE	00	2,768 K	
searchindexer.exe	SYSTEM	00	29,496 K	1
services.exe	SYSTEM	00	4,232 K	
smss.exe	SYSTEM	00	720 K	
SnagIt32.exe	Cem Kaner	04	25,456 K	
SnagPriv.exe	Cem Kaner	00	3,284 K	
SPM7.exe	Cem Kaner	00	7,668 K	
spoolsv.exe	SYSTEM	00	8,968 K	
stsystra.exe	Cem Kaner	00	8,992 K	
svchost.exe	SYSTEM	00	4,520 K	
svchost.exe	SYSTEM	00	5,912 K	
svchost.exe	NETWORK SERVICE	00	4,968 K	
svchost.exe	SYSTEM	00	30,848 K	
svchost.exe	NETWORK SERVICE	00	4,284 K	
svchost.exe	LOCAL SERVICE	00	5,540 K	
SynTPEnh.exe	Cem Kaner	00	5,032 K	
System	SYSTEM	00	240 K	
System Idle Process	SYSTEM	96	28 K	
taskmgr.exe	Cem Kaner	00	5,120 K	
tbkntservice.exe	SYSTEM	00	1,380 K	
tbksche.exe	SYSTEM	00	5,192 K	
TscHelp.exe	Cem Kaner	00	3,052 K	
vsmon.exe	SYSTEM	00	40,080 K	1
WindowsSearch.exe	Cem Kaner	00	19,532 K	
winlogon.exe	SYSTEM	00	1,548 K	
WLTRAY.EXE	Cem Kaner	00	8,356 K	
WLTRY5VC.EXE	SYSTEM	00	1,928 K	
zclient.exe	Cem Kaner	00	16,704 K	

Can you specify all of the possible outcomes?



Based on notes from Doug Hoffman

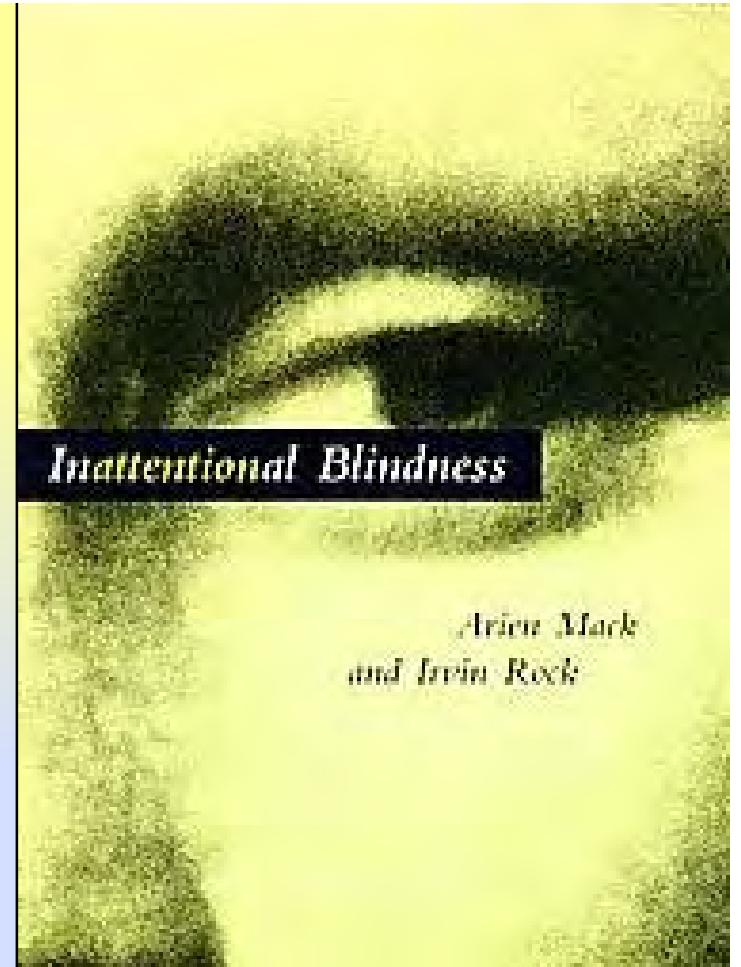
Next, A little demonstration...

- <http://www.geekarmy.com/Science/Crazy-Vision-Test.html>
- <http://www.dothetest.co.uk/>
- http://viscog.beckman.uiuc.edu/djs_lab/demos.html

Inattentional blindness

- Most (or all) people are subject to this:
 - Varying estimates of how many people fail to see the gorilla (etc.) in any particular experiment, but people who show no IB in one demonstration often miss the figure in the next
- What is important about inattentional blindness is NOT
 - Selective attention
 - ° (we've known about that for years and years and years)
- It is that IB demonstrates:

pre-attentive, semantically-based filtering (we ignore things based on their meaning, before we ever become aware of them).



Blind spots and unexpected outcomes

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

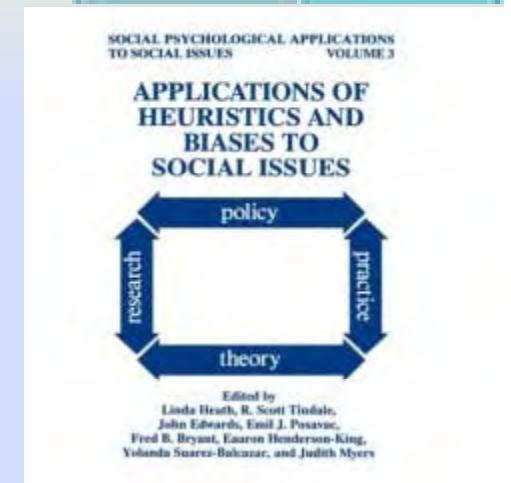
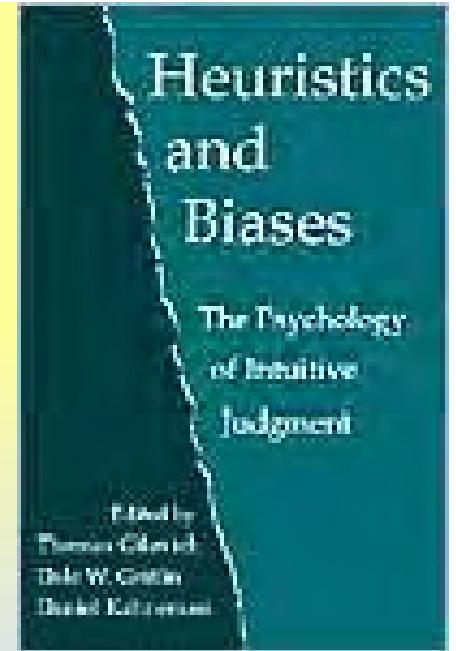
The 1100 embedded diagnostics

- 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*

Selective processing / biases

- Obama versus Clinton versus McCain
- Dartmouth / Princeton football demonstration
 - Hastorf, A. H. & Cantril, H. (1954). They saw a game: A case study. *Journal of Abnormal and Social Psychology*, 49, 129-134.
 - Smoker / Nonsmoker studies of confirmation bias
 - http://en.wikipedia.org/wiki/Confirmation_bias
- People will interpret what they see consistently with what they expect / want
 - Expected results drive expectancies
- If you set testers to believe they will find failures, they will find more failures and miss fewer ones



Scripts are hit and miss ...

People are finite capacity information processors

- We pay attention to some things
 - and therefore we do NOT pay attention to others
 - Even events that “should be” obvious will be missed if we are attending to other things.

Computers focus only on what they are programmed to look at:

- They are inattentionally blind by design

Scripts bias you to
miss the same things every time.

Generalize:

What if we focused on the cognitive,
persuasive, and other societal aspects of
software testing?
Would we learn more about testing?
About how to do the work of testing well?

Example 4

- Software engineering measurement deviates from theory of measurement in most other fields by underemphasizing construct validity (by not asking how we know whether the "measurement" actually measures what we are trying to measure). The result is predictable mischief, and the abandonment of metrics programs at most non-government-contractor companies.

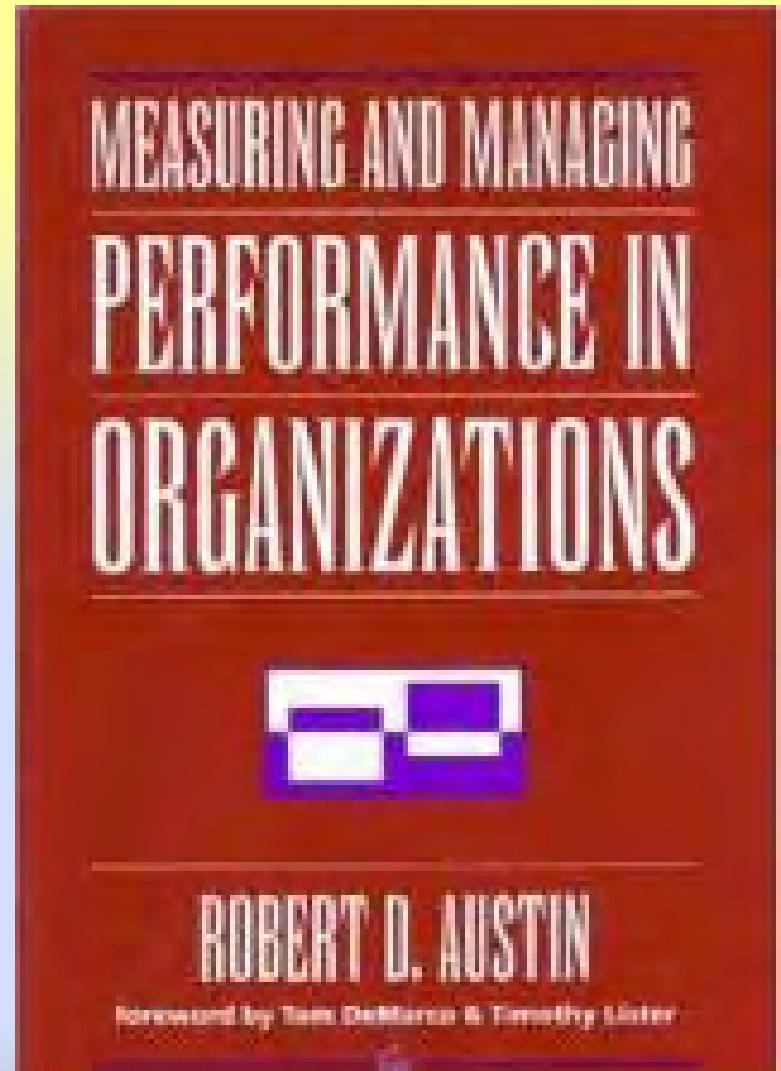
Test-related metrics

Most testing metrics are human performance metrics

- How productive is this tester?
- How good is her work?
- How good is someone else's work?
- How long is this work taking them?

These are well studied questions in the social sciences and not well studied when we ignore the humans and fixate on the computer.

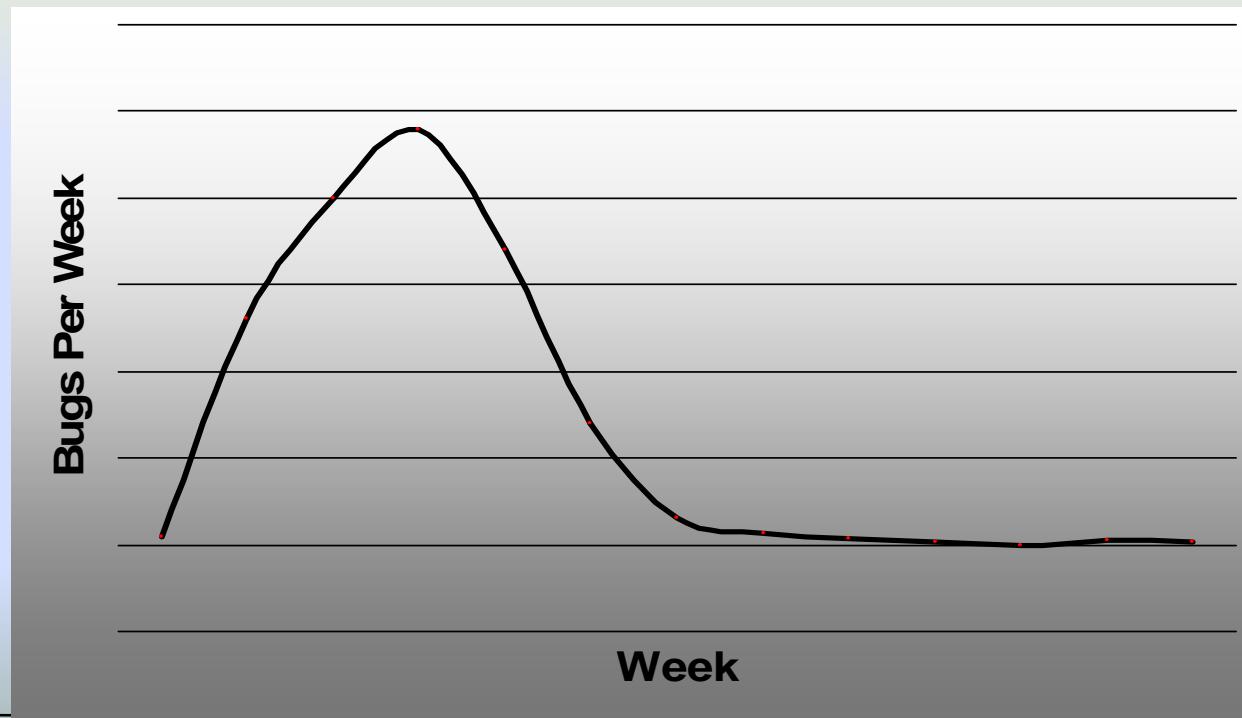
We ignore the human issues at risk.



Example: Bug find rates

Some people measure completeness of testing with bug curves:

- New bugs found per week ("Defect arrival rate")
- Bugs still open (each week)
- Ratio of bugs found to bugs fixed (per week)



Weibull reliability model

Bug curves can be useful progress indicators, but some people fit the data to theoretical curves to determine when the project will complete.

The model's assumptions

1. Testing occurs in a way similar to the way the software will be operated.
 2. All defects are equally likely to be encountered.
 3. Defects are corrected instantaneously, without introducing additional defects.
 4. All defects are independent.
 5. There is a fixed, finite number of defects in the software at the start of testing.
 6. The time to arrival of a defect follows the Weibull distribution.
 7. 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.
- See Erik Simmons, When Will We Be Done Testing? Software Defect Arrival Modeling with the Weibull Distribution.

The Weibull model

I think it's absurd to rely on a distributional model (or any model) when every assumption it makes about testing is obviously false.

One of the advocates of this approach points out that

“Luckily, the Weibull is robust to most violations.”

- This illustrates the use of surrogate measures—we don't have an attribute description or model for the attribute we really want to measure, so we use something else, that is allegedly “robust”, in its place. This can be very dangerous
- The Weibull distribution has a shape parameter that allows it to take a very wide range of shapes. If you have a curve that generally rises then falls (one mode), you can approximate it with a Weibull.

But how should we interpret an adequate fit to an otherwise indefensible model?

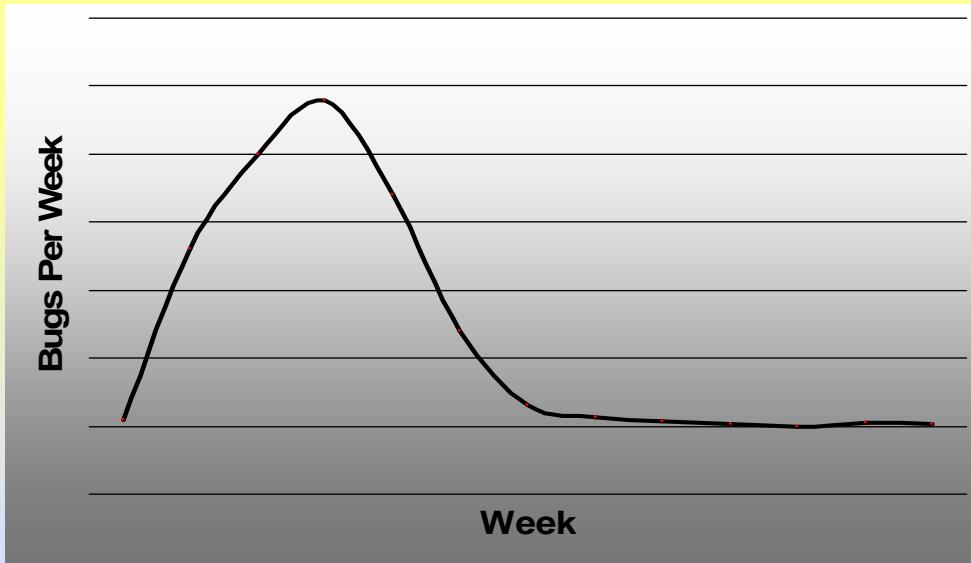
Side effects of bug curves

When development teams are pushed to show project bug curves that look like the Weibull curve, they are pressured to show a rapid rise in their bug counts, an early peak, and a steady decline of bugs found per week.

In practice, project teams, including testers, in this situation often adopt dysfunctional methods, doing things that will be bad for the project over the long run in order to make the numbers go up quickly.

- For more on measurement dysfunction, read Bob Austin's book, *Measurement and Management of Performance in Organizations*.
- For more observations of problems like these in reputable software companies, see Doug Hoffman's article, *The Dark Side of Software Metrics*.

Side effects of bug curves: Early testing



Predictions from these curves are based on parameters estimated from the data. You can start estimating the parameters once the curve has hit its peak and gone down a bit.

The sooner the project hits its peak, the earlier we would predict the product will ship.

So, early in testing, the pressure on testers is to drive the bug count up quickly, as soon as possible.

Side effects of bug curves

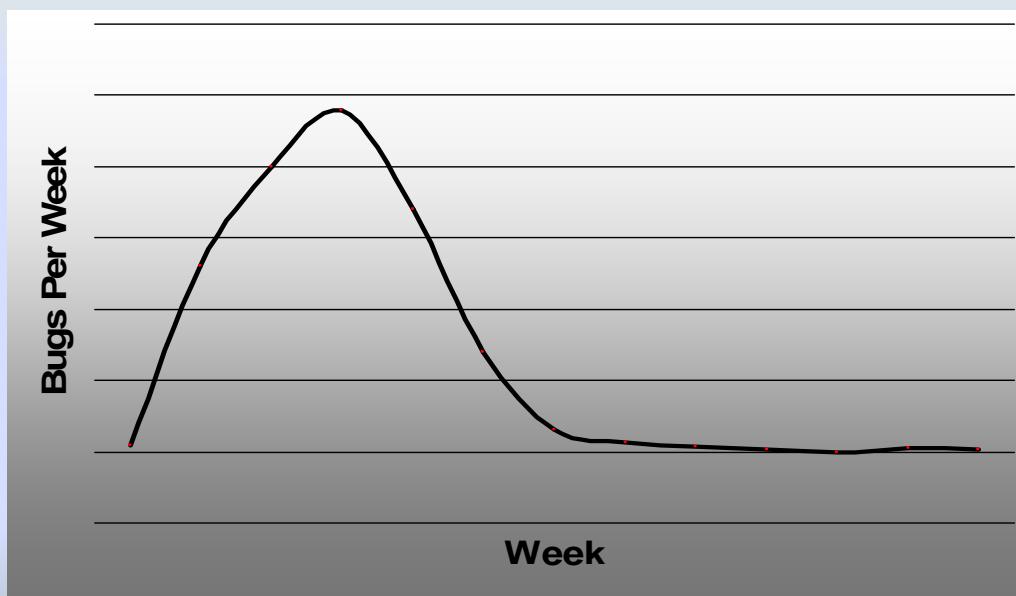
Earlier in testing, the pressure is to increase bug counts. In response, testers will:

- 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.)

Side effects of bug curves: Later in testing

After we get past the peak, the expectation is that testers will find fewer bugs each week than they found the week before.

Based on the number of bugs found at the peak, and the number of weeks it took to reach the peak, the model can predict the later curve, how many bugs per week in each subsequent week.



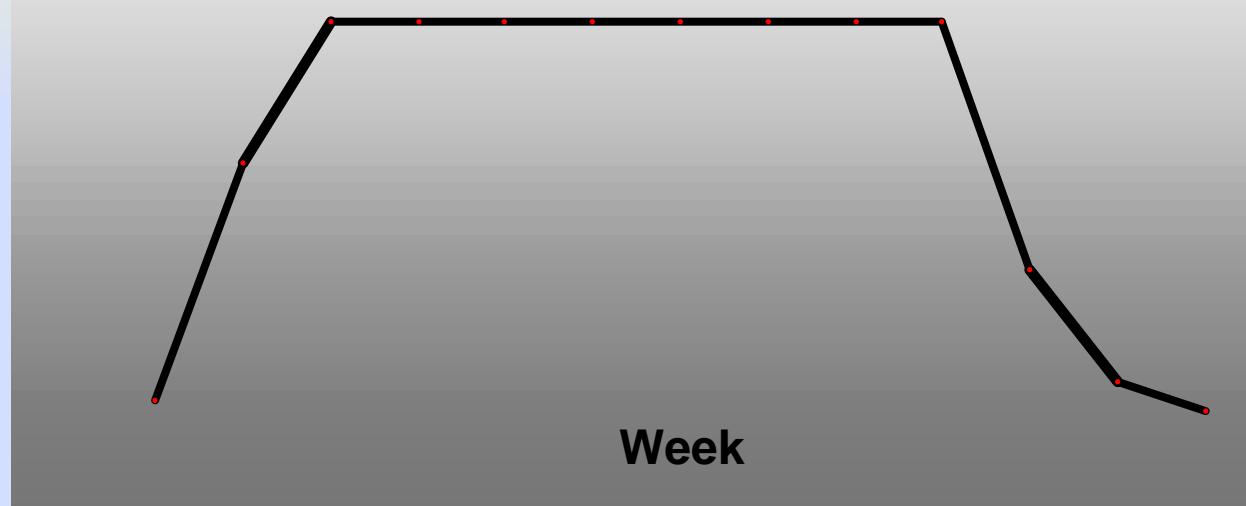
Side effects of bug curves

Later in testing, the pressure is to decrease the 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, keeping 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.

Bad models are counterproductive

*Shouldn't We Strive For
This ?*



Let's Sum Up

Is testing ONLY concerned with the human issues associated with product development and product use?

- Of course not
- But thinking in terms of the human issues leads us into interesting questions about
 - what tests we are running (and why)
 - what risks we are anticipating
 - how/why these risks are important, and
 - what we can do to help our clients get the information they need to manage the project, use the product, or interface with other professionals.

These are FUNDAMENTALLY DIFFERENT questions from the ALSO CRITICALLY IMPORTANT questions about implementation quality.

The tools are different, the concepts are different, the frames of reference are different, I don't know how to teach these (WELL) together in a single course.

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.