

# Approaches to Test Automation

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

Presentation at

Research in Motion

Kitchener/Waterloo, September 2009

Copyright (c) Cem Kaner 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 Grant 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.

# Acknowledgements

- This work was partially supported by NSF Grant EIA-0113539 ITR/SY+PE “Improving the education of software testers” and NSF Grant 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 authors and do not necessarily reflect the views of the National Science Foundation.
- Many of the ideas in this presentation were initially jointly developed with Doug Hoffman, as we developed a course on test automation architecture, and in the Los Altos Workshops on Software Testing (LAWST) and the Austin Workshop on Test Automation (AWTA).
  - LAWST 5 focused on oracles. Participants were Chris Agruss, James Bach, Jack Falk, David Gelperin, Elisabeth Hendrickson, Doug Hoffman, Bob Johnson, Cem Kaner, Brian Lawrence, Noel Nyman, Jeff Payne, Johanna Rothman, Melora Svoboda, Loretta Suzuki, and Ned Young.
  - LAWST 1-3 focused on several aspects of automated testing. Participants were Chris Agruss, Tom Arnold, Richard Bender, James Bach, Jim Brooks, Karla Fisher, Chip Groder, Elisabeth Hendrickson, Doug Hoffman, Keith W. Hooper, III, Bob Johnson, Cem Kaner, Brian Lawrence, Tom Lindemuth, Brian Marick, Thanga Meenakshi, Noel Nyman, Jeffery E. Payne, Bret Pettichord, Drew Pritsker, Johanna Rothman, Jane Stepak, Melora Svoboda, Jeremy White, and Rodney Wilson.
  - AWTA also reviewed and discussed several strategies of test automation. Participants in the first meeting were Chris Agruss, Robyn Brilliant, Harvey Deutsch, Allen Johnson, Cem Kaner, Brian Lawrence, Barton Layne, Chang Lui, Jamie Mitchell, Noel Nyman, Barindralal Pal, Bret Pettichord, Christiano Plini, Cynthia Sadler, and Beth Schmitz.
- We’re indebted to Hans Buwalda, Elisabeth Hendrickson, Noel Nyman, Pat Schroeder, Harry Robinson, James Tierney, & James Whittaker for additional explanations of test architecture and stochastic testing.
- We also appreciate the assistance and hospitality of “Mentsville,” a well-known and well-respected, but can’t-be-named-here, manufacturer of mass-market devices that have complex firmware. Mentsville opened its records to us, providing us with details about a testing practice (Extended Random Regression testing) that’s been evolving at the company since 1990.
- Finally, we thank Alan Jorgensen for explaining hostile data stream testing to us and providing equipment and training for us to use to extend his results.

# Two currently-fashionable approaches to test automation

- Unit (and low-level integration) testing using frameworks like xUnit and FIT (***programmer testing***)
- Regression testing via the user interface, often tied tightly to the specification (use case, story) (***customer "acceptance" testing***)

# People test for a lot of reasons:

Objective	Programmer tests	System level "acceptance" regression via the GUI
hunt for bugs	weak	weak
maintainability	good	weak, or counterproductive
insight into the internal design	good	minimal
insight into the implementation	good	minimal
insight into customer acceptability	weak	weak
demonstrate compliance to documented customer requirements	weak (unless requirements include glass-box verification)	strong in the same ways that very traditional heavyweight tests were strong -- but this is what Agile Manifesto was trying to take us away from, not lead us to...

## Two currently-fashionable approaches to test automation

- Unit (and low-level integration) testing using frameworks like xUnit and FIT (***programmer testing***)
- Regression testing via the user interface, often tied tightly to the specification (use case, story) (***customer "acceptance" testing***)

## Today's talk

- At the system level, all test automation is "computer assisted testing" rather than complete testing
- We have a wide range of choices for the types of assistance we select
- We have better choices than Specification-based (story-based) GUI regression testing for achieving most of the objectives of system-level testing
- I'll illustrate this with a family of examples of high-volume automated testing
- This is just ONE approach to better system-level automation

# The Telenova Station Set

1984. First phone on the market with an LCD display.

One of the first PBX's with integrated voice and data.

108 voice features, 110 data features. accessible through the station set



# The Telenova stack failure



July 4, 1985 12:01 PM Ext: 257  
Directory Admin Messages Voice Data

1-(212)662-7777 Connected Ext: 567  
Transfer Record Confernce Park Acct

Please enter selection  
LVMsa GetMsa Greeting Code

Ted K. waiting Wt:1 Hd:0  
I'llCall CallLater PlsWait Answ

Select a call & lift handset Wt:5 Hd:5  
Ted K. Peter T. Trunk 6 Trk 2Trk 7

Xenix 3 Connected for Data  
Transfer Baud EndCall Park Acct

Context-sensitive display  
10-deep hold queue  
10-deep wait queue

# The Telenova stack Failure -- A bug that triggered high-volume simulation

Beta customer (a stock broker) reported random failures

Could be frequent at peak times

- An individual phone would crash and reboot, with other phones crashing while the first was rebooting
- On a particularly busy day, service was disrupted all (East Coast) afternoon

We were mystified:

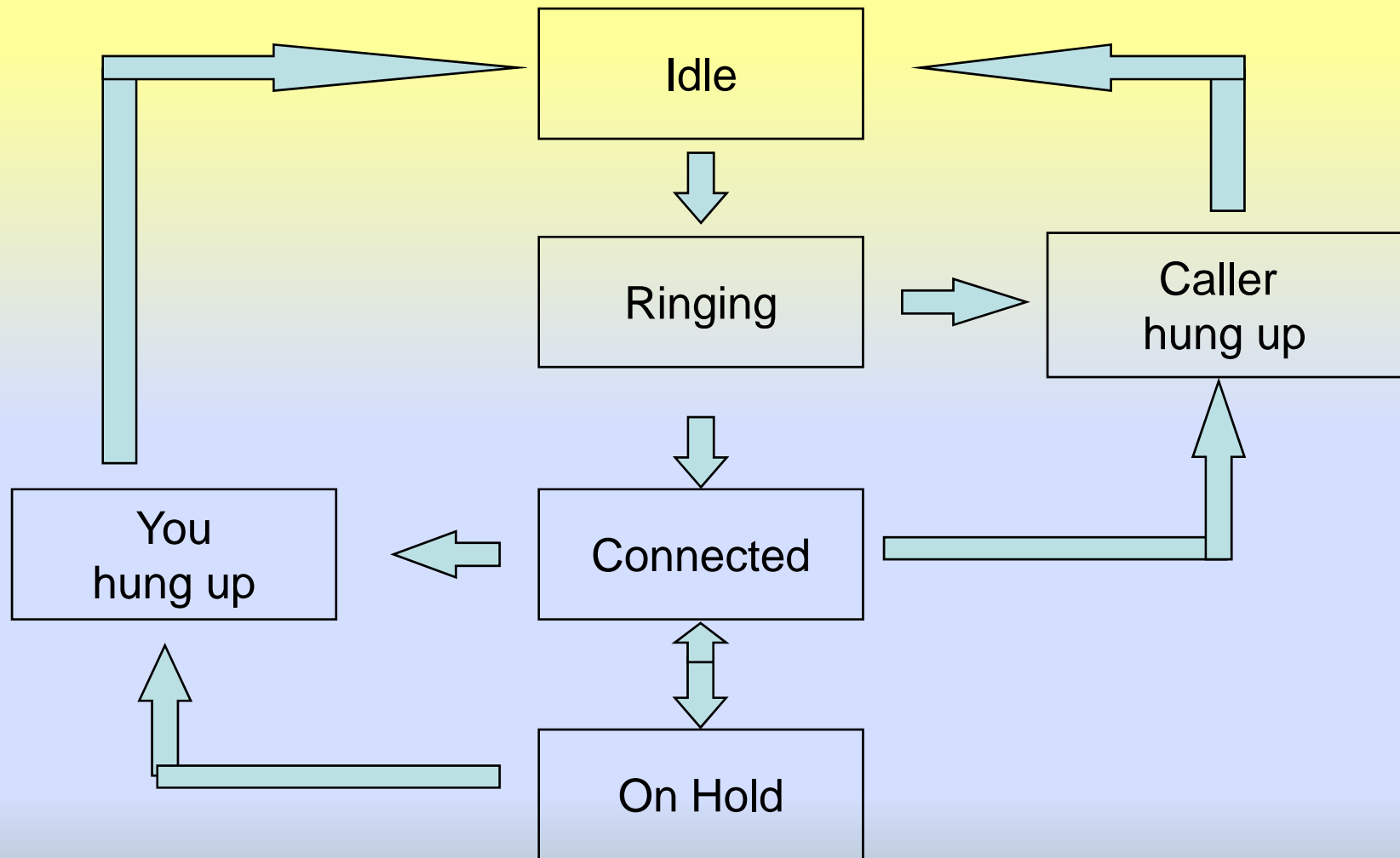
- All individual functions worked
- We had tested all lines and branches.

Ultimately, we found the bug in the hold queue

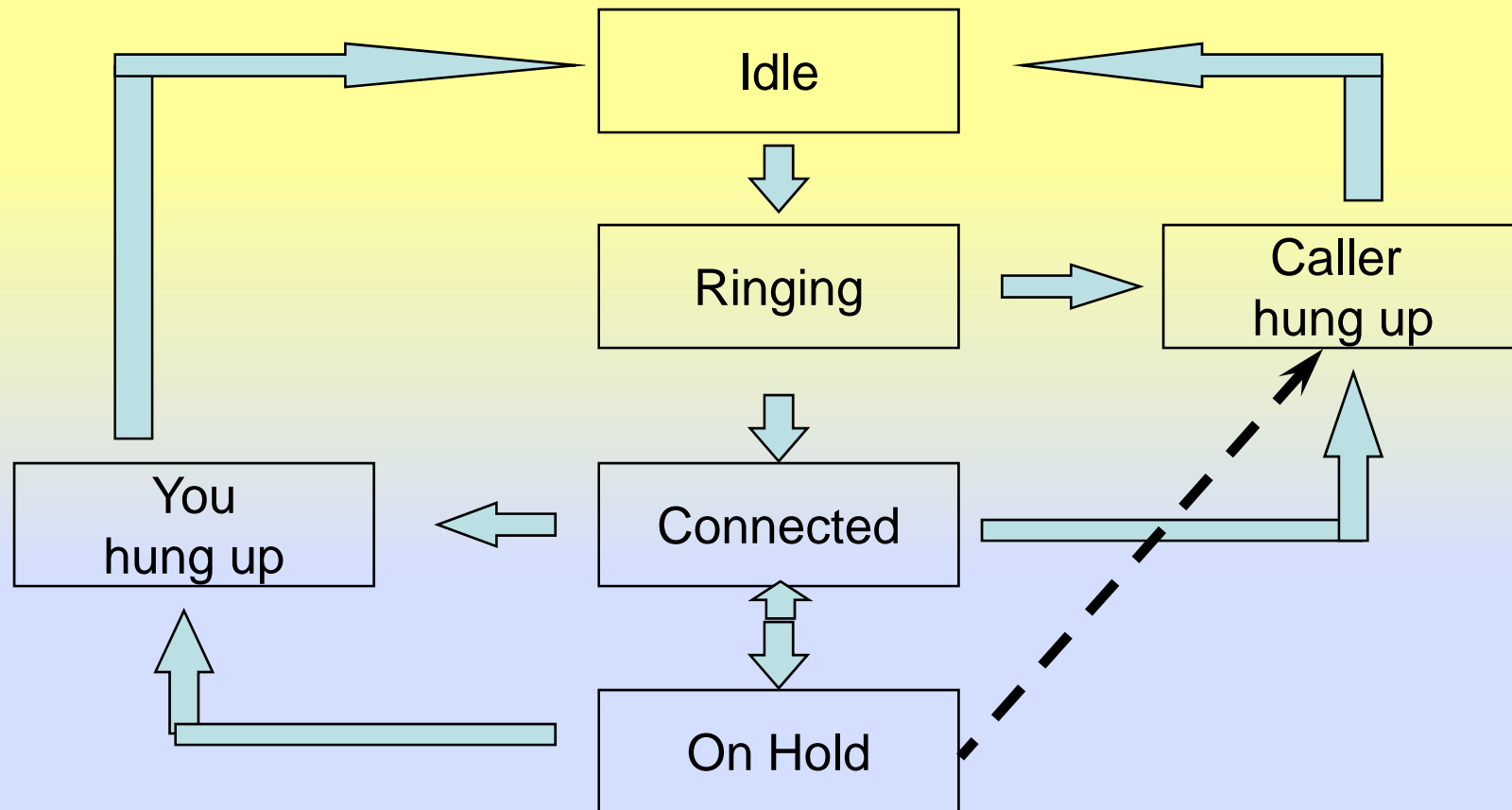
- Up to 10 calls on hold, each adds record to the stack
- Initially, the system checked stack whenever call was added or removed, but this took too much system time. So we dropped the checks and added these
  - Stack has room for 20 calls (just in case)
  - Stack reset (forced to zero) when we knew it should be empty
- The error handling made it almost impossible for us to detect the problem in the lab. Because we couldn't put more than 10 calls on the stack (unless we knew the magic error), we couldn't get to 21 calls to cause the stack overflow.



# The Telenova stack failure: A simplified state diagram showing the bug




## Telenova stack failure



When the caller hung up, we cleaned up everything *but* the stack. Failure was invisible until crash. From there, held calls were hold-forwarded to other phones, filling their held-call stacks, ultimately triggering a rotating outage.

# Telenova stack failure

**Having found and fixed  
the hold-stack bug,  
should we assume  
that we've taken care of the problem  
or that if there is one long-sequence bug,  
there will be more?**

**Hmmm   
If you kill a cockroach in your kitchen,  
do you assume  
you've killed the last bug?  
Or do you call the exterminator?**

# Simulator with probes



Telenova (\*) created a simulator

- generated long chains of random events, emulating input to the system's 100 phones
- could be biased, to generate more holds, more forwards, more conferences, etc.



Programmers added probes (non-crashing asserts that sent alerts to a printed log) selectively

- can't probe everything b/c of timing impact



After each run, programmers and testers tried to replicate failures, fix anything that triggered a message. After several runs, the logs ran almost clean.



At that point, shift focus to next group of features.



Exposed lots of bugs

(\*) By the time this was implemented, I had joined Electronic Arts.

# Telenova stack failure

- Simplistic approaches to path testing can miss critical defects.
- Critical defects can arise under circumstances that appear (in a test lab) so specialized that you would never intentionally test for them.
- Many of the failures probably corresponded to hard-to-reproduce bugs reported from the field.
  - These types of failures are hard to describe/explain in field reports
- When (in some future course or book) you hear a new methodology for combination testing or path testing:
  - test it against this defect.
  - If you had no suspicion that there was a stack corruption problem in this program, would the new method lead you to find this bug?

## A second case study: Long-sequence regression

- Welcome to “Mentenville”, a household-name manufacturer, widely respected for product quality, who chooses to remain anonymous.
- Mentenville applies wide range of tests to their products, including unit-level tests and system-level regression tests.
  - We estimate > 100,000 regression tests in “active” library
- Long-Sequence Regression Testing (LSRT)
  - Tests taken from the pool of tests ***the program has passed in this build.***
  - The tests sampled are run in random order until the software under test fails (e.g crash).
- Note that
  - these tests are no longer testing for the failures they were designed to expose.
  - these tests add nothing to typical measures of coverage

# Long-sequence regression testing

- Typical defects found include timing problems, memory corruption (including stack corruption), and memory leaks.
- Recent (2004) release: 293 reported failures exposed 74 distinct bugs, including 14 showstoppers.
- Mentsville's assessment is that *LSRT exposes problems that can't be found in less expensive ways*.
  - troubleshooting these failures can be very difficult and very expensive
  - wouldn't want to use LSRT for basic functional bugs or simple memory leaks--too expensive.
- LSRT has gradually become one of the fundamental techniques relied on by Mentsville
  - gates release from one milestone level to the next.

# High volume automated testing

These illustrate automated testing, but otherwise, they have little in common with the more widely discussed approaches that we call "automated testing"

- Programmer testing (always-automated unit tests and subsystem integration tests)
- System-level regression testing, typically at the GUI level

Let's look at these in turn...



Programmer Testing	System Testing
<ul style="list-style-type: none"> <li>• Does the program do what I intended?</li> <li>• Evidence is taken from the programmer's intent, which might be reflected in design documents, unit tests, comments, or personal memory</li> <li>• 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</li> <li>• Tools: Unit test frameworks (e.g. JUNIT), code coverage, complexity metrics, version control, source code analyzers, state models</li> </ul>	<ul style="list-style-type: none"> <li>• Does the program meet the needs of the stakeholders?</li> <li>• 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.)</li> <li>• 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.</li> <li>• Tools are diverse. GUI regression tests are common but wasteful. More useful to think in terms of computer-assisted testing.</li> <li>• High volume tools are in infancy, but vital</li> </ul>

Programmer Testing	System Testing
<ul style="list-style-type: none"> <li>• All programmers do programmer testing to some degree. Even weak programmers find the vast majority of their own bugs (public vs private bugs)</li> <li>• This IS programming. This helps the programmer understand her implementation (or the implementation by a colleague).</li> <li>• The tools are easy. What to DO with the tools is hard: <ul style="list-style-type: none"> <li>• <b><u>Problem decomposition</u></b></li> <li>• Discrete math (including graphs)</li> <li>• Boolean logic (complex combinations)</li> </ul> </li> </ul>	<ul style="list-style-type: none"> <li>• About 20% to 60% of the new product development effort (in terms of staff size)</li> <li>• 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)</li> </ul>

# Similar names for fundamentally different approaches

Programmer testing: Test-first (test-driven) development	System testing: Test then code ("proactive testing")
The programmer creates 1 test, writes code, gets the code working, refactors, moves to next test	The <i>tester</i> creates many tests and then the <i>programmer</i> codes
Primarily unit tests and low-level integration	Primarily acceptance, or system-level tests
Near-zero delay, communication cost	Usual process inefficiencies and delays (code, then deliver build, then wait for test results, slow, costly feedback)
Supports exploratory development of architecture, requirements, & design	Supports understanding of requirements
Widely discussed, fundamental to XP, not so widely adopted	Promoted as a "best practice" for 30 years, recently remarketed as "agile" system testing

# Test-driven development

- Provides a structure for working from examples, rather than from an abstraction. (Supports a common learning / thinking style.)
- Provides concrete communication with future maintainers.
- Provides a unit-level regression-test suite (change detectors)
  - support for refactoring
  - support for maintenance
- Makes bug finding / fixing more efficient
  - No roundtrip cost, compared to GUI automation and bug reporting.
  - No (or brief) delay in feedback loop compared to external tester loop
- Provides support for experimenting with the component library or language features

# Unit testing can spare us from simplistic system testing #1

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

See Hunt & Thomas, *Pragmatic Unit Testing*, for examples of unit test design. These are usually confirmatory tests.

- Imagine testing a method that sorts a list:
  - Try a maximum value in the middle of the list, check that it appears at the end of the list
  - Try a huge value
  - Try a maximum length list
  - Try a null value
  - Try a value of wrong type
  - Try a negative value
  - Try a value that should sort to the start of the list.
  - Exact middle of the list
  - Exercise every error case in the method
  - Try a huge list
  - Try a max+1 length list
  - Insert into a null list
  - Try a tied value
  - Try a zero?

# Unit testing can spare us from simplistic system testing #2

- If the programmers do thorough unit testing
  - Based on their own test design, or
  - Based on a code analyzer / test generator (like Agitator)
- then apart from a sanity-check sample at the system level, we don't have to repeat these tests as system tests.
- Instead, we can focus on techniques that exercise the program more broadly and more interestingly

# Unit testing can spare us from simplistic system testing #3

- Many testing books treat domain testing (boundary / equivalence analysis) as **the** primary system testing technique. To the extent that it teaches us to do risk-optimized stratified sampling when we deal with a large space of tests, domain testing offers powerful guidance.
- But the specific technique—checking single variables and combinations at their edge values—is often handled well in unit and low-level integration tests. These are more efficient than system tests.
- If the programmers actually test this way, then system testers should focus on other risks and other techniques.
- When other people do an honest and serious job of testing in their way, a system test group so jealous of its independence that it refuses to consider what has been done by others is bound to waste time repeating simple tests and thereby miss opportunities to try more complex tests focused on harder-to-assess risks.

# Typical system-level testing tasks

- Analyze product & its risks
  - market
  - benefits & features
  - review source code
  - platform & associated software
- Develop testing strategy
  - pick key techniques
  - prioritize testing foci
- Design tests
  - select key test ideas
  - create test for the idea
- Run test first time (often by hand)
- Evaluate results
  - Report bug if test fails
- Keep archival records
  - trace tests back to specs
- Manage testware environment
- **If we create regression tests:**
  - Capture or code steps once test passes
  - Save “good” result
  - Document test / file
  - Execute the test
    - Evaluate result
      - » Report failure or
      - » Maintain test case



# Automating system-level testing tasks

No testing tool covers this range of tasks

In automated regression testing:

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

"GUI-level automated system testing" doesn't mean  
*automated testing*

"GUI-level automated system testing" means  
*computer-assisted testing*

- So, the key design question is, where do we need the most assistance?

# What other computer-assistance would be valuable

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

# Regression testing

- We do regression testing in order to check whether problems that the previous round of testing would have exposed have come into the product in this build.
- We are NOT testing to confirm that the program "still works correctly"
  - It is impossible to completely test the program, and so
    - we never know that it "works correctly"
    - we only know that we didn't find bugs with our previous tests
- A regression test series:
  - has relatively few tests
    - tests tied to stories, use cases, or specification paragraphs can be useful but there are not many of them. They do not fully explore the risks of the product.
  - every test is lovingly handcrafted (or should be) because we need to maximize the value of each test

# Regression testing

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

- 5000 regression tests?
- 5000 tests the program has passed over and over.
- Should we try some other tests instead?

# Cost/benefit the system regression tests

## COSTS?

- Maintenance of UI / system-level tests is not free
  - change the design of the program
  - discover an inconsistency between the new program and the test
  - discover the problem is obsolescence of the test
  - change the test

## BENEFITS?

- What information will we obtain from re-use of this test?
- What is the value of that information?
- How much does it cost to automate the test the first time?
- How much maintenance cost for the test over a period of time?
- How much inertia does the maintenance create for the project?
- How much support for rapid feedback does the test suite provide for the project?

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

# The concept of inertia

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

The less inertia we build into a project, the more responsive the development group can be to stakeholder requests for change (design changes and bug fixes).

- Intentional inertia:
  - Change control boards
  - User interface freezes
- Process-induced inertia: Costs of change imposed by the development process
  - rewrite the specification
  - maintenance costs of tests
  - execution costs of regression tests
- Reduction of inertia is usually seen as a core objective of agile development.

# Cost / benefit of system-level regression

- To reduce costs and inertia
- And maximize the information-value of our tests
- Perhaps we should concentrate efforts on reducing our UI-level regression testing rather than trying to automate it
  - Eliminate redundancy between unit tests and system tests
  - Develop high-volume strategies to address complex problems
  - Repeat system level tests less often:
    - risk-focused regression rather than procedural
    - explore new scenarios rather than reusing old ones
      - » scenarios give us information about the product's design, but once we've run the test, we've gained that information. A good scenario test is not necessarily a good regression test
    - create a framework for specifying new tests easily, interpreting the specification, and executing the tests programmatically

# Risk-focused rather than procedural regression testing

- Procedural regression
  - Do the same test over and over (reuse same tests each build)
- Risk-focused regression
  - Check for the same risks each build, but use different tests (e.g. combinations of previous tests)
  - See [www.testingeducation.org/BBST/BBSTRegressionTesting.html](http://www.testingeducation.org/BBST/BBSTRegressionTesting.html)
- However,
  - The risk-focused tests are different every build and so traditional GUI regression automation is an unavailable strategy



# Back to high-volume automation: 11 examples

1. simulator with probes
2. long-sequence regression
3. function equivalence testing
4. comparison to a computational or logical model
5. comparison to a heuristic predictor, such as prior behavior
6. state-transition testing without a state model (dumb monkeys)
7. state-transition testing using a state model (terminate on failure rather than on a preset coverage criterion)
8. functional testing in the presence of high background load
9. hostile data stream testing
10. random inputs to protocol checkers
11. combination tests with extensive or exhaustive combination of values of N variables

# A Structure for Thinking about HVAT

## INPUTS

- What is the source for our inputs? How do we choose input values for the test?
- (“Input” includes the full set of conditions of the test)

## OUTPUTS

- What outputs will we observe?

## EVALUATION

- How do we tell whether the program passed or failed?

## EXPLICIT MODEL?

- Is our testing guided by any explicit model of the software, the user, the process being automated, or any other attribute of the system?

## WHAT ARE WE MISSING?

- The test highlights some problems but will hide others.

## SEQUENCE OF TESTS

- Does / should any aspect of test N+1 depend on test N?

## THEORY OF ERROR

- What types of errors are we hoping to find with these tests?

## TROUBLESHOOTING SUPPORT

- What data are stored? How else is troubleshooting made easier?

## BASIS FOR IMPROVING TESTS?

## HOW TO MEASURE PROGRESS?

- How much, and how much is enough?

## MAINTENANCE LOAD / INERTIA?

- Impact of / on change to the SUT

## CONTEXTS

- When is this useful?

# Appendices

We probably run out of time here.

- Appendix 1 elaborates the examples of high volume automated testing
- Appendix 2 considers the objectives of system testing

I provide added text with these, for use as post-talk reference material.

If we have time left, I'll probably skip to the system testing (Appendix 2) because it is more likely to provoke more discussion.

# Appendix I: High volume automated testing

# Back to high-volume automation: 11 examples

1. simulator with probes
2. long-sequence regression
3. function equivalence testing
4. comparison to a computational or logical model
5. comparison to a heuristic predictor, such as prior behavior
6. state-transition testing without a state model (dumb monkeys)
7. state-transition testing using a state model (terminate on failure rather than on a preset coverage criterion)
8. functional testing in the presence of high background load
9. hostile data stream testing
10. random inputs to protocol checkers
11. combination tests with extensive or exhaustive combination of values of N variables

# Simulator with probes

## INPUTS:

- Random, but with biasable transition probabilities.

## OUTPUTS

- Log messages generated by the probes. These contained some troubleshooting information (whatever the programmer chose to include).

## EVALUATION STRATEGY

- Read the log, treat any event leading to a log message as an error.

## EXPLICIT MODEL?

- At any given state, the simulator knows what the SUT's options are, but it doesn't verify the predicted state against actual state.

## WHAT ARE WE MISSING?

- Any behavior other than log

## SEQUENCE OF TESTS

- Ongoing sequence, never reset.

## THEORY OF ERROR

- Long-sequence errors (stack overflow, memory corruption, memory leak, race conditions, resource deadlocks)

## TROUBLESHOOTING SUPPORT

- Log messages

## BASIS FOR IMPROVING TESTS?

- Clean up logs after each run by eliminating false alarms and fixing bugs. Add more tests and log details for hard-to-repro errors

# Mentsville LSRT

## INPUTS:

- taken from existing regression tests, which were designed under a wide range of criteria

## OUTPUTS

- Mentsville: few of interest other than diagnostics
- Others: whatever outputs were interesting to the regression testers, plus diagnostics

## EVALUATION STRATEGY

- Mentsville: run until crash or other obvious failure
- Others: run until crash or until mismatch between program behavior or prior results or model predictions

## EXPLICIT MODEL?

- None

## WHAT ARE WE MISSING?

- Mentsville: Anything that doesn't cause a crash
- Others: Anything that doesn't cause a crash OR a mismatch with the predicted result of the current regression test

## SEQUENCE OF TESTS

- LSRT sequencing is random

## THEORY OF ERROR

- bugs not easily detected by the regression tests: long-fuse bugs, such as memory corruption, memory leaks, timing errors

## TROUBLESHOOTING SUPPORT

- diagnostics log, showing state of system before and after tests

# NEXT: Function equivalence testing

Classic example: Doug Hoffman (2003) Exhausting your test options, Software Testing & Quality Engineering magazine, July/August 2003, p. 10-11  
([www.softwarequalitymethods.com/Papers/Exhaust%20Options.pdf](http://www.softwarequalitymethods.com/Papers/Exhaust%20Options.pdf))

Example from a recent final exam in Florida Tech's *Testing 2* :

- Use test driven development to create a test tool that will test the Open Office spreadsheet by comparing it with Excel
- (We used COM interface for Excel and an equivalent interface for OO, drove the API-level tests with a program written in Ruby, a simple scripting language)
- Pick 10 functions in OO (and Excel). For each function:
  - Generate random input to the function
  - Compare OO evaluation and Excels
  - Continue until you find errors or are satisfied of the equivalence of the two functions.
- Now test expressions that combine several of the tested functions



# Function equivalence testing

## INPUTS:

- Random

## OUTPUTS

- We compare output with the output from a reference function. In practice, we also independently check a small sample of calculations for plausibility

## EVALUATION STRATEGY

- Output fails to match, or fails to match within delta, or testing stops from crash or other obvious misbehavior.

## EXPLICIT MODEL?

- The reference function is, in relevant respects, equivalent to the software under test.
- If we combine functions (testing expressions rather than single functions), we need a grammar or other basis for describing combinations.

## WHAT ARE WE MISSING?

- Anything that the reference function can't generate

## SEQUENCE OF TESTS

- Tests are typically independent

## THEORY OF ERROR

- Incorrect data processing / storage / calculation

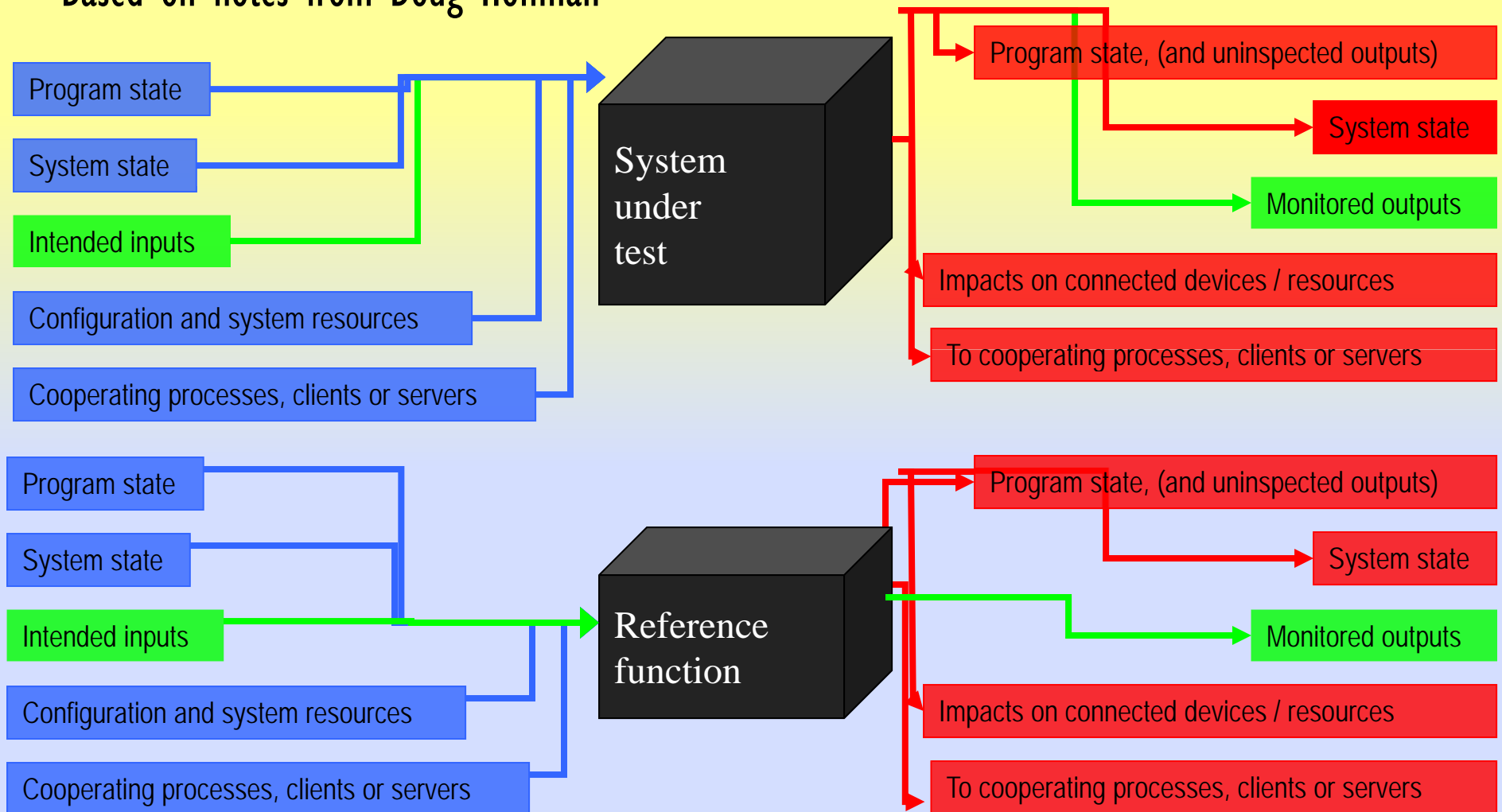
## TROUBLESHOOTING SUPPORT

- Inputs saved

## BASIS FOR IMPROVING TESTS?

# What do you compare, when you use an oracle?

Based on notes from Doug Hoffman



# Can you specify your test configuration?

Comparison to a reference function is fallible.  
We only control some inputs and observe some results (outputs).

For example, do you know whether the test and reference systems are equivalently configured?

- 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?

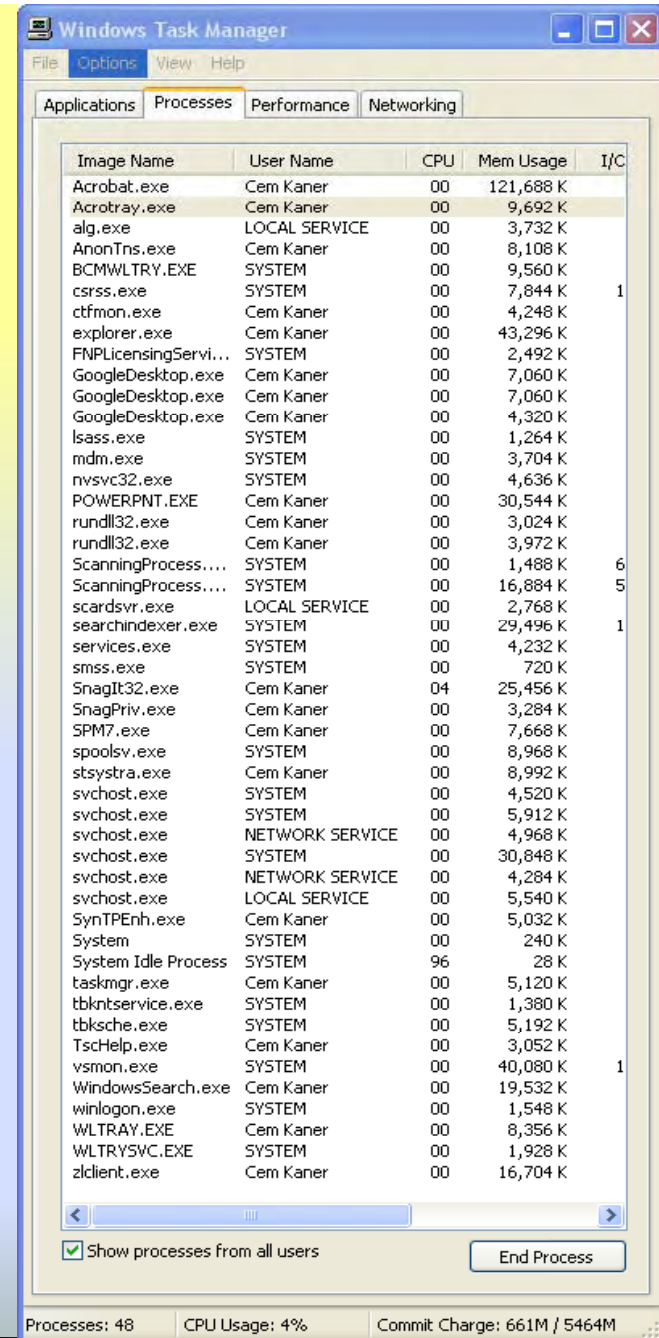


Image Name	User Name	CPU	Mem Usage	I/O
Acrobat.exe	Cem Kaner	00	121,688 K	
Acrotray.exe	Cem Kaner	00	9,692 K	
alg.exe	LOCAL SERVICE	00	3,732 K	
AnonTns.exe	Cem Kaner	00	8,108 K	
BCMFWLTRY.EXE	SYSTEM	00	9,560 K	
csrss.exe	SYSTEM	00	7,844 K	1
ctfmon.exe	Cem Kaner	00	4,248 K	
explorer.exe	Cem Kaner	00	43,296 K	
FNPLicensingServi...	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	
nsvsc32.exe	SYSTEM	00	4,636 K	
POWERPNT.EXE	Cem Kaner	00	30,544 K	
rundll32.exe	Cem Kaner	00	3,024 K	
rundll32.exe	Cem Kaner	00	3,972 K	
ScanningProcess....	SYSTEM	00	1,488 K	6
ScanningProcess....	SYSTEM	00	16,884 K	5
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	
SnagitPriv.exe	Cem Kaner	00	3,284 K	
SPM7.exe	Cem Kaner	00	7,668 K	
spoolsv.exe	SYSTEM	00	8,968 K	
stsysstra.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 Idle Process	SYSTEM	00	240 K	
taskmgr.exe	Cem Kaner	00	5,120 K	
tbkntservice.exe	SYSTEM	00	1,380 K	
tbksche.exe	SYSTEM	00	5,192 K	
TschHelp.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	
WLTRYVX.EXE	Cem Kaner	00	8,356 K	
WLTRYVXVX.EXE	SYSTEM	00	1,928 K	
zclient.exe	Cem Kaner	00	16,704 K	

☒ Show processes from all users End Process

Processes: 48 CPU Usage: 4% Commit Charge: 661M / 5464M

# Comparison to a computational or logical model

## INPUTS:

- Random or systematic

## OUTPUTS

- We compare output with the output from a model. For example, we might check a function by inverting it (e.g. square the value obtained by a square root function under test)

## EVALUATION STRATEGY

- Output fails to match, or fails to match within delta, or testing stops from crash or other obvious misbehavior.

## EXPLICIT MODEL?

- The reference model is, in relevant respects, equivalent to the software under test.

## WHAT ARE WE MISSING?

- Anything that the reference model can't generate. For example, we have no predictions about memory management

## SEQUENCE OF TESTS

- Tests are typically independent

## THEORY OF ERROR

- Incorrect data processing / storage / calculation

## TROUBLESHOOTING SUPPORT

- Inputs saved

## BASIS FOR IMPROVING TESTS?

# Comparison to a heuristic predictor

A heuristic is a fallible idea or method that may you help simplify and solve a problem.

Heuristics can hurt you when used as if they were authoritative rules.

Heuristics may suggest wise behavior, but only in context. They do not contain wisdom.

Your relationship to a heuristic is the key to applying it wisely.

“Heuristic reasoning is not regarded as final and strict but as provisional and plausible only, whose purpose is to discover the solution to the present problem.”

- George Polya, *How to Solve It*

## 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 (p. 70) 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”

# Some useful oracle heuristics

***Consistent within product:*** Function behavior consistent with behavior of comparable functions or functional patterns within the product.

***Consistent with comparable products:*** Function behavior consistent with that of similar functions in comparable products.

***Consistent with history:*** Present behavior consistent with past behavior.

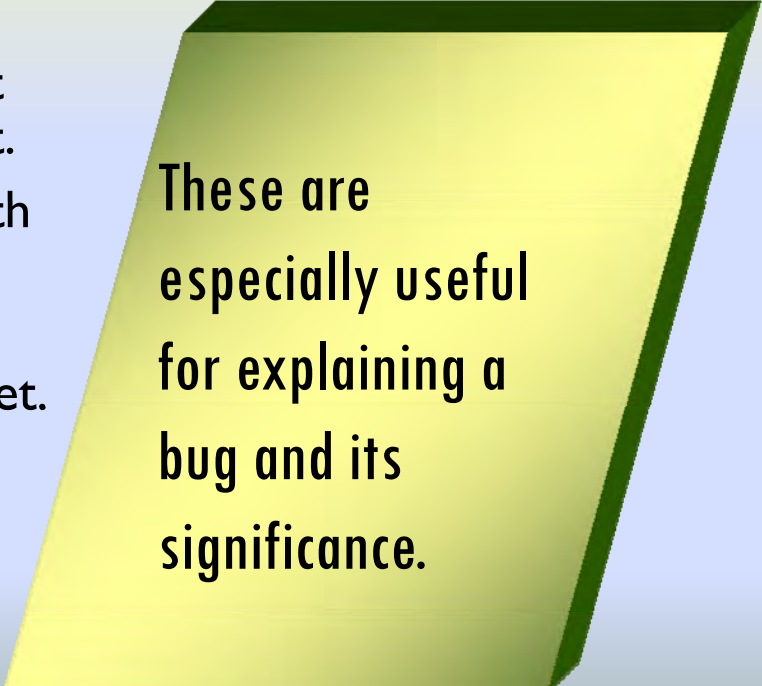
***Consistent with our image:*** Behavior consistent with an image the organization wants to project.

***Consistent with claims:*** Behavior consistent with documentation or ads.

***Consistent with specifications or regulations:*** Behavior consistent with claims that must be met.

***Consistent with user's expectations:*** Behavior consistent with what we think users want.

***Consistent with Purpose:*** Behavior consistent with product or function's apparent purpose.



These are  
especially useful  
for explaining a  
bug and its  
significance.

# State transition testing

State transition testing is *stochastic*. It helps to distinguish between independent random tests and stochastic tests.

## Random Testing

- Random (or statistical or stochastic) testing involves generating test cases using a random number generator. Individual test cases are not optimized against any particular risk. The power of the method comes from running large samples of test cases.

## Independent Random Testing

- Our interest is in each test individually, the test before and the test after don't matter.

## Stochastic Testing

- A stochastic process involves a series of random events over time
  - Stock market is an example
  - Program may pass individual tests when run in isolation: The goal is to see whether it can pass a large series of the individual tests.



## State transition tests without a state model: Dumb monkeys

- Phrase coined by Noel Nyman. Many prior uses (UNIX kernel, Lisa, etc.)
- Generate a long sequence of random inputs driving the program from state to state, but without a state model that allows you to check whether the program has hit the correct next state.
  - **Executive Monkey:** (dumbest of dumb monkeys) Press buttons randomly until the program crashes.
  - **Clever Monkey:** No state model, but knows other attributes of the software or system under test and tests against those:
    - Continues until crash or a diagnostic event occurs. The diagnostic is based on knowledge of the system, not on internals of the code. (Example: button push doesn't push—this is system-level, not application level.)
    - Simulator-with-probes is a clever monkey
- Nyman, N. (1998), "Application Testing with Dumb Monkeys," *STAR West*.
- Nyman, N. "In Defense of Monkey Testing," <http://www.softtest.org/sigs/material/nnyman2.htm>

# Dumb monkeys

## INPUTS:

- Random generation.
- Some commands or parts of system may be blocked (e.g. format disk)

## OUTPUTS

- May ignore all output (executive monkey) or all but the predicted output.

## EVALUATION STRATEGY

- Crash, other blocking failure, or mismatch to a specific prediction or reference function.

## EXPLICIT MODEL?

- None

## WHAT ARE WE MISSING?

- Most output. In practice, dumb monkeys often lose power quickly (i.e. the program can pass it even though it is still full of bugs).

## SEQUENCE OF TESTS

- Ongoing sequence, never reset

## THEORY OF ERROR

- Long-sequence bugs
- Specific predictions if some aspects of SUT are explicitly predicted

## TROUBLESHOOTING SUPPORT

- Random number generator's seed, for reproduction.

## BASIS FOR IMPROVING TESTS?

## State transitions: State models (smart monkeys)

- For any state, you can list the actions the user can take, and the results of each action (what new state, and what can indicate that we transitioned to the correct new state).
  - Randomly run the tests and check expected against actual transitions.
  - See [www.geocities.com/model\\_based\\_testing/online\\_papers.htm](http://www.geocities.com/model_based_testing/online_papers.htm)
  - The most common state model approach seems to drive to a level of coverage, use Chinese Postman or other algorithm to achieve all sequences of length N. (A lot of work along these lines at Florida Tech)
    - ***High volume approach runs sequences until failure appears or the tester is satisfied that no failure will be exposed.***
  - Coverage-oriented testing fails to account for the problems associated with multiple runs of a given feature or combination.
- 
- Al-Ghafees, M. A. (2001). Markov Chain-based Test Data Adequacy Criteria. Unpublished Ph.D., Florida Institute of Technology, Melbourne, FL. Summary at <http://ecommerce.lebow.drexel.edu/eli/2002Proceedings/papers/AlGha180Marko.pdf>
  - Robinson, H. (1999a), "Finite State Model-Based Testing on a Shoestring," *STAR Conference West*. Available at [www.geocities.com/model\\_based\\_testing/shoestring.htm](http://www.geocities.com/model_based_testing/shoestring.htm).
  - Robinson, H. (1999b), "Graph Theory Techniques in Model-Based Testing," *International Conference on Testing Computer Software*. Available at [www.geocities.com/model\\_based\\_testing/model-based.htm](http://www.geocities.com/model_based_testing/model-based.htm).
  - Whittaker, J. (1997), "Stochastic Software Testing", *Annals of Software Engineering*, 4, 115-131.

# State-model based testing

## INPUTS:

- Random, but guided or constrained by a state model

## OUTPUTS

- The state model predicts values for one or more reference variables that tell us whether we reached the expected state.

## EVALUATION STRATEGY

- Crash or other obvious failure.
- Compare to prediction from state model.

## EXPLICIT MODEL?

- Detailed state model or simplified model: *operational modes*.

## WHAT ARE WE MISSING?

- The test highlights some relationships and hides others.

## SEQUENCE OF TESTS

- Does any aspect of test N+1 depend on test N?

## THEORY OF ERROR

- Transitions from one state to another are improperly coded
- Transitions from one state to another are poorly thought out (we see these at test design time, rather than in execution)

## TROUBLESHOOTING SUPPORT

- What data are stored? How else is troubleshooting made easier?

## BASIS FOR IMPROVING TESTS?

# Functional testing in the presence of high background load

- Alberto Savoia (2000, May 1-5, 2000). The science and art of web site load testing. Paper presented at the International Conference on Software Testing Analysis & Review, Orlando, FL.

This is an example of a paper demonstrating a rise in functional failures as system load increases. In his data, as load rose above 50% of capacity, failure rates began increasing dramatically (a hockey stick function) from very low levels.

# Hostile data stream testing

- Pioneered by Alan Jorgensen (FIT, recently retired)
- Take a “good” file in a standard format (e.g. PDF)
  - corrupt it by substituting one string (such as a really, really huge string) for a much shorter one in the file
  - feed it to the application under test
  - Can we overflow a buffer?
- Corrupt the “good” file in thousands of different ways, trying to distress the application under test each time.
- Jorgenson and his students showed serious security problems in some products, primarily using brute force techniques.
- Method seems appropriate for application of genetic algorithms or other AI to optimize search.

# Hostile data streams and HVAT

## INPUTS:

- A series of random mutations of the base file

## OUTPUTS

- Simple version--not of much interest

## EVALUATION STRATEGY

- Run until crash, then investigate

## EXPLICIT MODEL?

- None

## WHAT ARE WE MISSING?

- Data corruption, display corruption, anything that doesn't stop us from further testing

## SEQUENCE OF TESTS

- Independent selection (without repetition). No serial dependence.

## THEORY OF ERROR

- What types of errors are we hoping to find with these tests?

## TROUBLESHOOTING SUPPORT

- What data are stored? How else is troubleshooting made easier?

## BASIS FOR IMPROVING TESTS?

- Simple version: hand-tuned
- Seemingly obvious candidate for GA's and other AI

# Combination tests with extensive or exhaustive combination of values of N variables

Suppose you have 5 variables that can each take 4 values of interest. There are  $4 \times 4 \times 4 \times 4 \times 4 = 1024$  possible tests. Many of the discussions of domain testing focus on a sampling strategy to test a small subset of these 1024 tests. Similarly, combinatorial sampling (such as all-pairs) can reduce the test set significantly (all pairs would require only 20 tests).

However, much recent academic research looks at large random samplings (or exhaustive sampling) from the set of N-tuples (a test is specified by one value for each of N variables)



## Appendix 2: Notes on system testing

# 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 or development for customers
- With evolving requirements (and therefore an incomplete and non-authoritative specification).

**Verification only begins to address the critical question:**

*Will this software meet user 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)

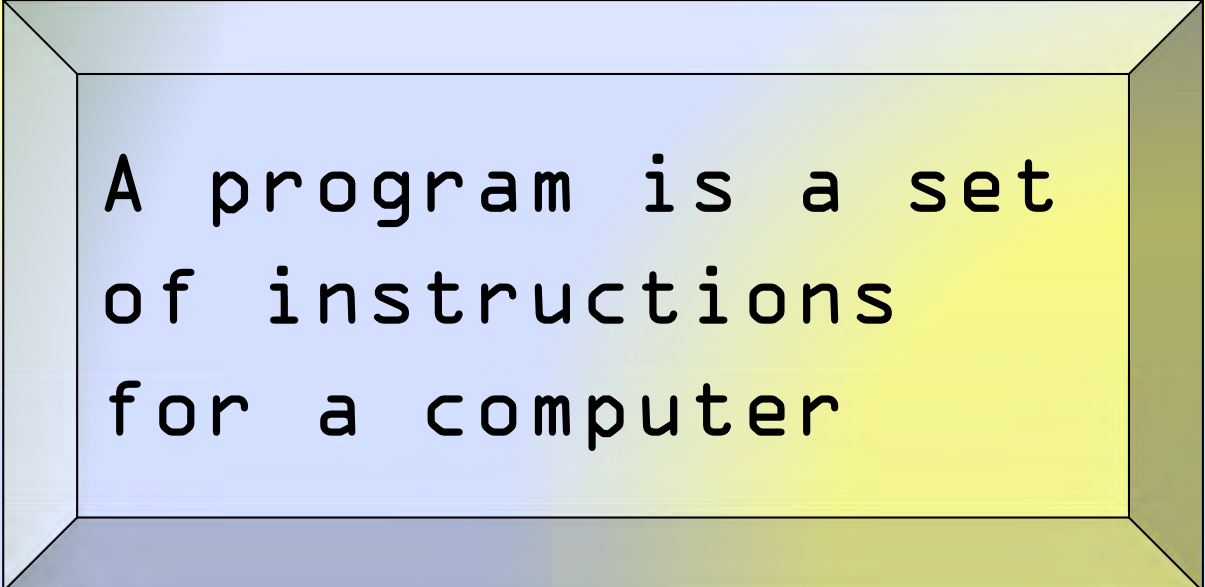
Designing system tests is like doing a requirements analysis. They rely on similar information but use it differently.

- 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.
- ***What are the economics of automating some or all of these tests?***

*A system  
tester's view of  
the world*

# What's a Computer Program?

Textbooks often define a “computer program” like this:



A program is 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.

*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.

# 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.*

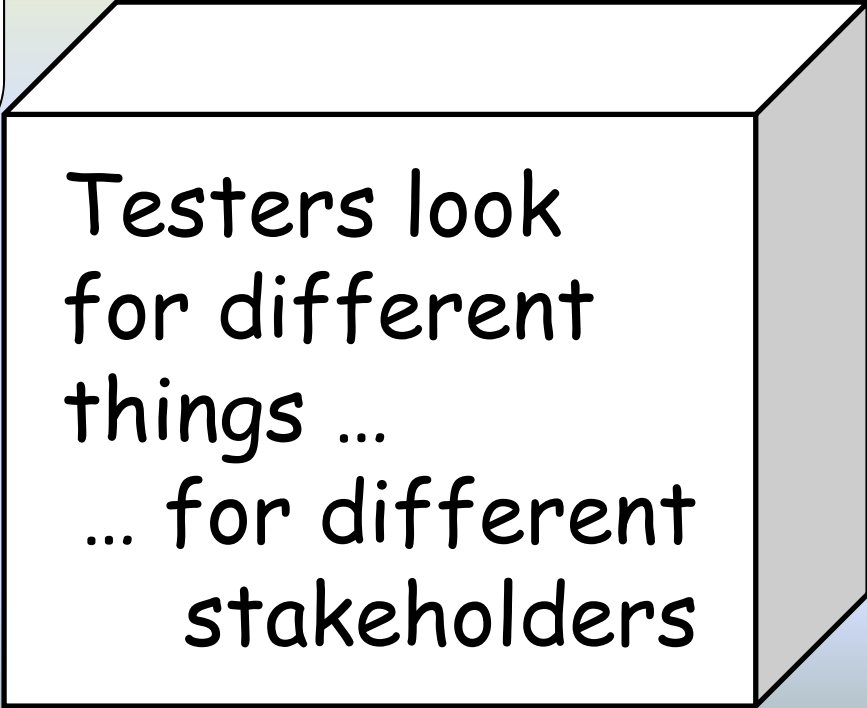
# What are we really testing for?

Under this view:

Quality is value to some person

-- Jerry Weinberg

- 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

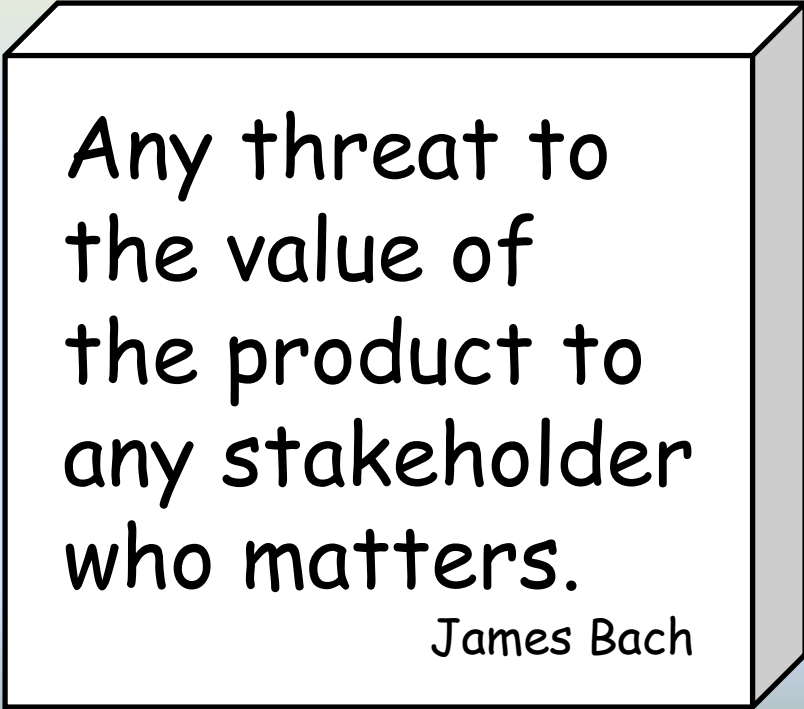
# Software error

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

A 3D box with a white front face and gray side faces. The front face contains a quote in a large, black, sans-serif font. The quote is: "Any threat to the value of the product to any stakeholder who matters." Below the quote, in a smaller font, is the name "James Bach".

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

James Bach

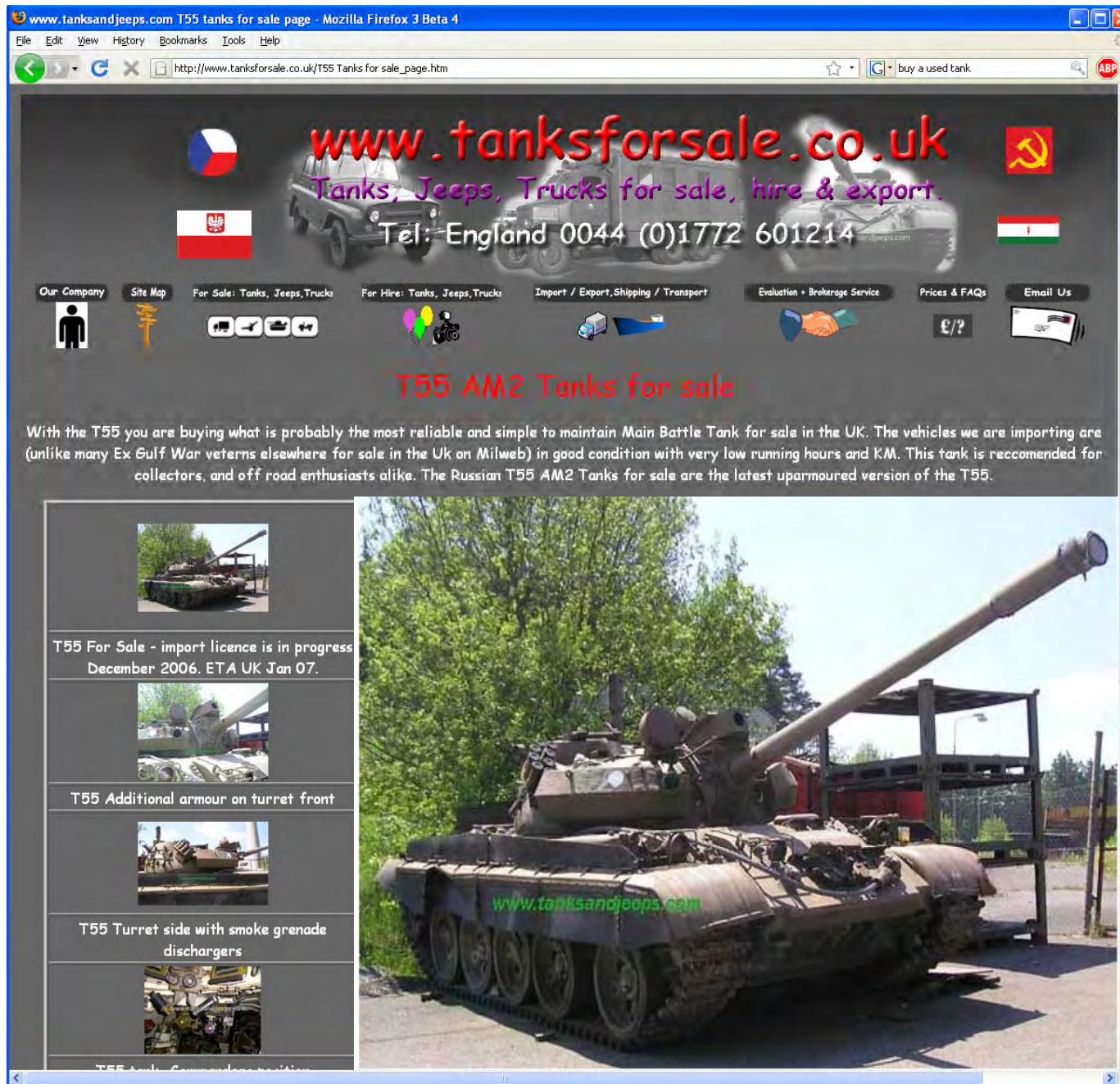
# What are we really testing for?

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?





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

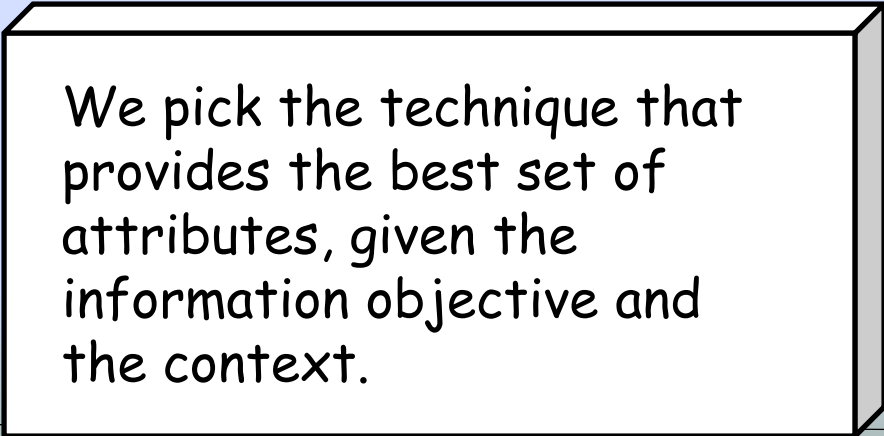
Different objectives require different testing tools and strategies and will yield different tests, different test documentation and different test results.



# 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.

# 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

# 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

# Scenario testing

- Story-based tests are usually based on sketchily-documented user stories. They are a type of scenario test.

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.

# 16 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 the specific transactions that people try to complete, such as opening a bank account or sending a message. What are all the steps, data items, outputs, displays, etc.?
8. What forms do the users work with? Work with them (read, write, modify, etc.)

# 16 ways to create good scenarios

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?

# Risks of scenario testing

Other approaches are better for testing early, unstable code.

- A scenario is complex, involving many features. If the first feature is broken, the rest of the test can't be run. Once that feature is fixed, the next broken feature blocks the test.
- Test each feature in isolation before testing scenarios, to efficiently expose problems as soon as they appear.

Scenario tests are not designed for coverage of the program.

- It takes exceptional care to cover all features or requirements in a set of scenario tests. Statement coverage simply isn't achieved this way.



# Risks of scenario testing

Reusing scenarios may lack power and be inefficient

- Documenting and reusing scenarios seems efficient because it takes work to create a good scenario.
- Scenarios often expose design errors but we soon learn what a test teaches about the design.
- Scenarios expose coding errors because they combine many features and much data. To cover more combinations, we need new tests.
- Do regression testing with single-feature tests or unit tests, not scenarios.