Software Testing as a Quality Improvement Activity

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

Lockheed Martin / IEEE Computer Society Webinar Series

September 3, 2009

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.

Abstract

Testing is often characterized as a relatively mindless activity that should be formalized, standardized, routinized, endlessly documented, fully automated, and most preferably, eliminated.
This webinar presents the contrasting view: good testing is challenging. cognitively complex, and customized to suit the circumstances of the individual project. The webinar presents testing as an empirical, technical investigation of a software product or service, conducted to provide quality-related information to stakeholders.

A fundamental challenge of testing is that cost/benefit analysis underlies every decision. Two tests are distinct if one can reveal an error that the other would miss. The population of distinct tests of any nontrivial program is infinite. And so any decision to do X is also a decision to not do the things that could have been done if the resources hadn't been spent on X. The question is not whether an activity is worthwhile. It is whether this activity is so much more worthwhile than the others that it would be a travesty not to do it (or at least, a little bit of it). For example, system-level regression-test automation might allow us to run the same tests thousands of times at low cost, but after the first few repetitions, how much do we learn from the typical regression test? What if instead, we spent the regression-test resources on new tests, addressing new risks (other ways the program could fail)?
Quality is not quantity. If our measure is amount (or value) of quality-related information for the stakeholders, what improves efficiency? Do we cover more ground by running in place very quickly, or by moving forward more slowly? Under what conditions do which types of automation improve testing effectiveness or efficiency?

Another fundamental challenge of testing is that quality is subjective—as Weinberg put it, "Quality is value to some person." Meeting a specification might trigger someone's duty to pay for a program, but if the program doesn't actually meet their needs, preferences, and expectations, they won't like it, won't want to use it, and certainly won't recommend it. How should we design a testing effort to expose the potential dissatisfiers for each stakeholder?

What's a Computer Program?

Textbooks often define a "computer program" like this:

> A program is a set of instructions for a computer

That's like defining a house

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

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

This focuses on stakeholders and purpose, rather than on the machine.

- This narrow focus on the machine prepares our students to make the worst errors in software engineering.

Something built for people to live in...

The focus is on

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

Stakeholder:

Any person affected by:

- success or failure of a project, or
- actions or inactions of a product, or
- 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?

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 (aka bug)

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, or a functional error

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

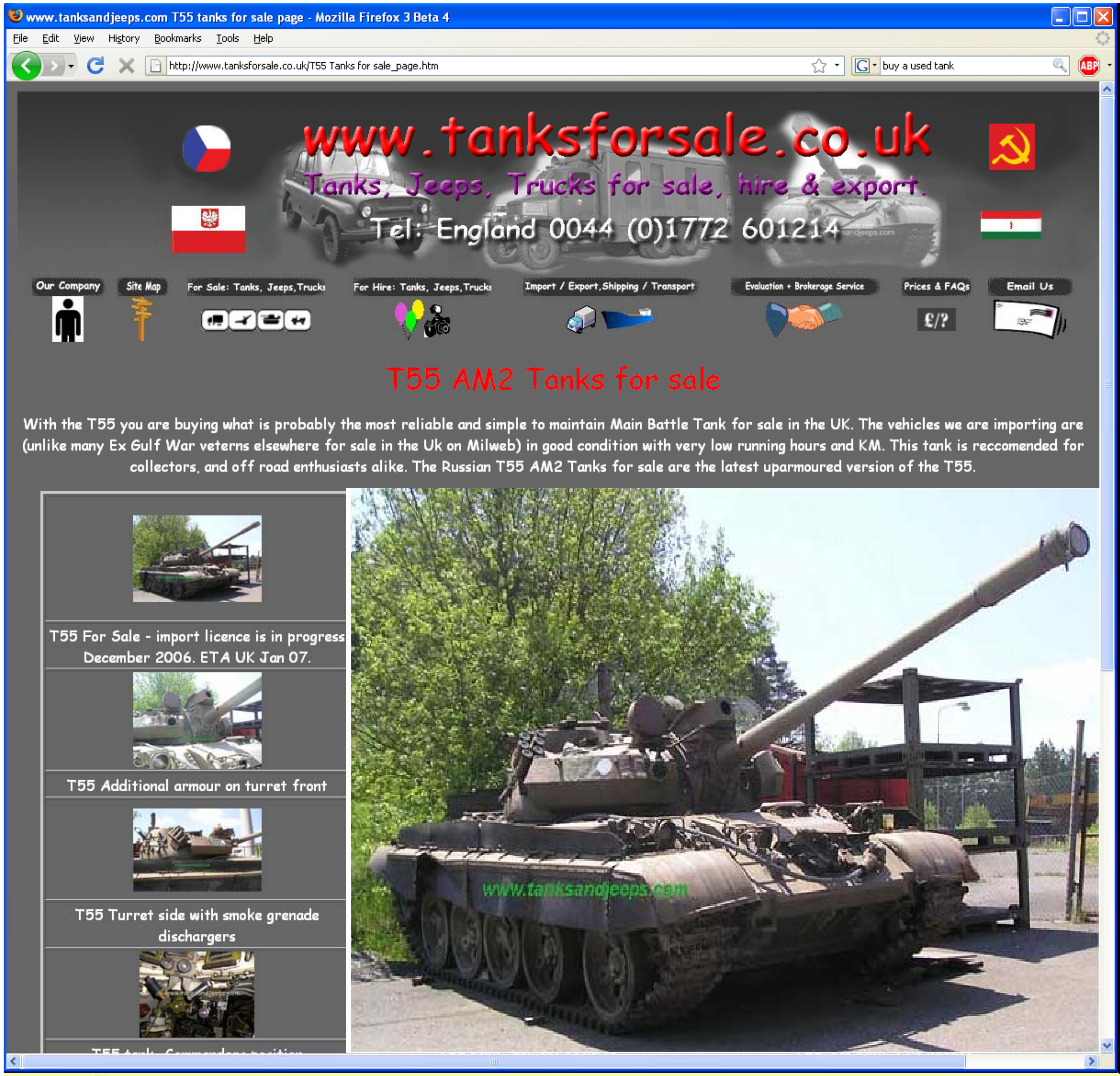
James Bach

Countervailing benefit?

American crash standards say that a car is defective if it gets damaged in a head-on 5 mph collision.

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

We DO know how to make motor vehicles that can withstand collisions at much higher speeds.


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
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
£/?
www.tanksandjeeps.com

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 veterns elsewhere for sale in the Uk on Milweb) in good condition with very low running hours and KM. This tank is reccomended 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

Not every limitation on value is a bug.

Effective bug reporting requires judgment, based on evaluation of the product's context and appropriate constraints within that context.

Examples of context factors that drive and constrain testing

- 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

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

Empirical? -- All tests are experiments.
Information? -- Reduction of uncertainty. Read Karl Popper (Conjectures & Refutations) on the goals of experimentation

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 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
- Device compatibility testing
- Testing to maximize statement and branch coverage

We pick the technique that provides the best set of attributes, given our context and the objective of our search

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

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

Example: Scenario testing

The ideal scenario has several characteristics:

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

Example: Scenario testing

- Scenarios tell compelling stories about the way the product (directly or indirectly) affects people.
- When a well-designed scenario fails, we understand why we should care that it failed.
- **Don't confuse scenarios with instantiated use cases.**
  - **The use-case definition replaces people with "actors" and goals and consequences with sequence diagrams.**
  - **It is a sterilized reframing of an established concept. For testers, the original one is the one that guides us to more compelling reports.**
  - **For the more traditional notions of "scenario", read the work of John M. Carroll and his colleagues.**

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?

(Video lecture at www.testingeducation.org/BBST)

16 ways to create good scenarios

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.)
9. Interview users about famous challenges and failures of the old system.

16 ways to create good scenarios

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.

16 ways to create good scenarios

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?

- Each of these ways is its own vector -- its own direction for creating a significant series of distinct tests.
- Each test in one of these families carries a lot of information about the design and value of the product.
- How much value (how much new information) should we expect from rerunning one of these tests?

A misguided analogy

- Software testing is often compared to manufacturing quality control.
- This is a fundamentally misleading analogy.
- We should be comparing our work to **design quality control**.

Manufacturing vs Design QC

Manufacturing QC	Design QC
• Individual instances might be defective	• If ANY instance is defective then EVERY instance is defective
• Manufacturing errors reflect deviation from an understood and intended characteristic (e.g. physical size).	• Variability is not the essence of defectiveness. Design errors reflect what we did NOT understand about the product. We don't know what they are. Error is building the wrong thing, not building it slightly wrongly.

Manufacturing QC	Design QC
• Finite set of types of manufacturing defects. We can standardize tests for each.	• Two tests are distinct if one can expose an error that the other would miss. Infinite set of possible distinct tests. We need to optimize for breadth of conditions tested, not standardization on a few conditions.
• Optimize for precision in description in test design and execution	• Optimize for variation that can provide new information. We need to discover new problems, not hunt for things already found.

Manufacturing QC	Design QC
• Repetition (test each instance the same way) carries information because some instances are defective (mis-manufactured) and others not.	• Repetition carries little information. (Let's focus on new iterations, not new copies): • In 1950's-1980's, errors used to randomly reappear in code. We eliminate the main causes by using version control systems. • The regression test suite is a collection of tests the program passed in previous versions. How much new information can you expect from that?

Manufacturing QC	Design QC
• Automation improves precision, reduces variability and lowers cost	• <u>Regression</u> automation often increases costs, reduces variability and thereby reduces flow of new information.

Back to scenarios

- Scenarios are a terrific technique for testing the design of a product.
- But they are terrible for manufacturing QC (for repetitive reuse)
  - too complex, makes them poorly optimized for isolating the cause of the failure
  - too complex, drives up maintenance costs in the face of frequent code changes / design changes in development
  - too focused on "real use" instead of optimal conditions for exposing a targeted failure.
- Every test technique has strengths and weaknesses. We need a diverse set, not a favorite.

System testing (validation)

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

- Requirements analysts try to foster agreement about the system to be built. Testers exploit disagreements to predict problems with the system.
- Testers don't have to decide how the product *should* work. Their task is to expose credible concerns to the stakeholders.
- Testers don't have to make product design tradeoffs. They expose *consequences* of those tradeoffs, especially unanticipated or serious consequences.
- The tester doesn't have to respect prior design agreements.
- The system tester's work cannot be exhaustive, just useful

Validation, Accreditation, Verification

- Validation:
  - Will this software meet our needs?
- Accreditation:
  - Should I certify this software as adequate for our needs?
- Verification (at the system level)
  - Does this software conform to documented assertions?
- In contract-driven development, verification can tell you that the client is now legally obligated to pay for the software.
- Verification cannot tell you whether the software will meet stakeholder needs or elicit a positive reaction from users.

Validation, Accreditation, Verification

- **At the system level**,
  - verification is about confirming that you are testing what you think you are testing
  - validation and accreditation are about evaluating the quality of the product

A longstanding absurdity

Is the bad-software problem really caused by bad requirements definition, which we could fix by doing a better job up front, if only we were more diligent and more professional in our work?

- We have made this our primary excuse for bad software for decades.
  - If this was really the problem, and if processes focusing on early lockdown of requirements provided the needed solution, wouldn't we have solved this by now?

A process that requires us to lock down decisions early will maximize our risk, not manage it.

A longstanding absurdity (2)

- We learn **throughout the project** about what is needed, what is useful, what is possible, what is cost-effective, what is high-risk.
  - Shouldn't we design our development process to take advantage of this growth of knowledge?
  - A process that requires us to lock down decisions early
    - causes us to make decisions at time of greatest ignorance
    - maximizes our risk.

A longstanding absurdity (3)

- "Chaos metrics" report how "badly" projects fare against their original requirements, budgets and schedule.
  - That's only chaos if you are fool enough to ***believe*** the original requirements, budgets and schedule.
  - We need to manage this, and quit whining about it.
  - (And quit paying consultants to tell us to whine about it.)

Alternatives?

- Alternative lifecycles:
  - Delay many "requirements" decisions and design decisions
  - Do several iterations of decide-design-code-test-fix-test, each building on the foundation laid by the one before.
    - Evolutionary development, RUP, came long before the Agile Manifesto

Agile Development

- The core of Agile development is:
  - optimization to support change throughout the lifecycle
- rather than
  - optimization to support project management around a fixed-price / fixed-feature contracting model
- "Embrace change"
  - processes are intended to
    - increase velocity (our ability to make change quickly)
    - decrease inertia (factors that drive down speed or drive up cost of change)

Programmer testing in the Agile World

- Programmer testing is resurrected as the primary area of focus
  - Many in the agile community reject independent system testing as a primary tool for quality control is flatly rejected as slow, expensive, and ineffective.
  - Their vision, which I firmly do not advocate here, is of a stripped-down "acceptance testing" that is primarily a set of automated regression tests designed to verify implementation of customer "stories" (essentially, brief-description use cases).
- Let's look instead at the contrast between programmer testing and the more robust system testing

Programmer Testing	System Testing
• 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	• 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.)

Programmer Testing	System Testing
• 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	• Tests are typically behavioral. For practical reasons they are usually black box (a subspecies of behavioral).
• Tools: Unit test frameworks (e.g. JUNIT), code coverage, complexity metrics, version control, source code analyzers, state models	• Tools are diverse. GUI regression tests are common but wasteful. More useful to think in terms of computer-assisted testing. • High volume test tools are in infancy, but vital

Programmer Testing	System Testing
• All programmers do programmer testing to some degree. Even weak programmers find the vast majority of their own bugs (public vs private bugs)	• About 20% to 60% of the new product development effort (in terms of staff size)
• This IS programming. This helps the programmer understand her implementation (or the implementation by a colleague).	• This is NOT primarily about programming. To a very large degree, this is applied social science, plus specific subject matter expertise.

Programmer Testing	System Testing
• Using the test tools is easy. • Current books and training overemphasize the implementation issues and underemphasize the underlying issues of test design. • We need more on WHAT to implement, rather than so much about HOW.	• Current books overemphasize management issues (practitioner books) or mathematical models (academic books) • We need more on how to imagine potential failure and consequences, optimize tests to trigger the failures of interest, tie these to stakeholder concerns. On the technical site, how to design simulators and other foundations for introducing massive variation into testing.

Programmer Testing: Test-driven development

- Decompose a desired application into parts. For each part:
  - Goal: Create a relatively simple implementation the meets the design objectives -- and works.
  - Code in iterations. In each iteration
    - decompose the part we are developing today into little tasks
    - write a test (that is, write a specification by example) for the "first" task
    - write simple code to extend the current code base in ways that will pass the new test while continuing to pass the previous tests
    - get the code working and clean (refactor it), then next iteration
- To create next build: integrate new part, then check integration success with full set of unit tests, plus the relevant system tests.

Test-driven development

- 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

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:
  - Anecdotal evidence (including my own observations of students) suggests that maintainers new to code base will learn the code faster from a good library of unit tests than from javadocs.
    - The unit tests become an interactive specification that support experimentation with the code (learning theory: constructivist, inquiry-based)

Unit testing can spare us from simplistic system testing

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

Example (next slide) -- consider common cases for testing a sort

- Little to be gained by reinventing these cases all the time
- These are easy for the programmer to code (assuming he has enough time)
- These are easy to keep in a regression test suite, and easy to keep up to date

Unit testing can spare us from simplistic system testing

- 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 huge list
- Try a maximum length list
- Try a max+1 length list
- Try a null value
- Insert into a null list
- Try a value of wrong type
- Try a tied value
- Try a negative value
- Try a zero?
- Try a value that should sort to the start of the list.
- Exact middle of the list
- Exercise every error case in the method

Unit testing can spare us from simplistic system testing

- 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

- Example: Many testing books treat domain testing (boundary / equivalence analysis) as **the** primary system testing technique.
- This 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.
- Beware the system test group so jealous of its independence that it squanders opportunities for complex tests focused on harder-to-assess risks by insisting on repeating simple tests already done by others.

Similar names for fundamentally different approaches

Test-first (test-driven) development Programmer testing:	Test then code ("proactive testing") "Acceptance" testing
The programmer creates 1 test, writes code, gets the code working, refactors, moves to next test	The *tester* creates many tests and *then* the *programmer* codes
Primarily unit tests and low-level integration	Primarily 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, but recent surveys (Dr. Dobbs) suggest it is not widely adopted in agile community	Promoted as a "best practice" for 30 years, recently remarketed as "agile" system testing, still ineffective (IMHO)

Automating system testing

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" results
  - Document the test & test data
  - **Execute saved tests**
  - Evaluate results
  - Maintenance on saved tests

What have we automated?
EXECUTION of OLD tests?
Is that the best we can do?

Automating system-level testing tasks

No testing tool covers this entire 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) or the maintenance (which is often VERY expensive).

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

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

Other computer-assistance?

- 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.)

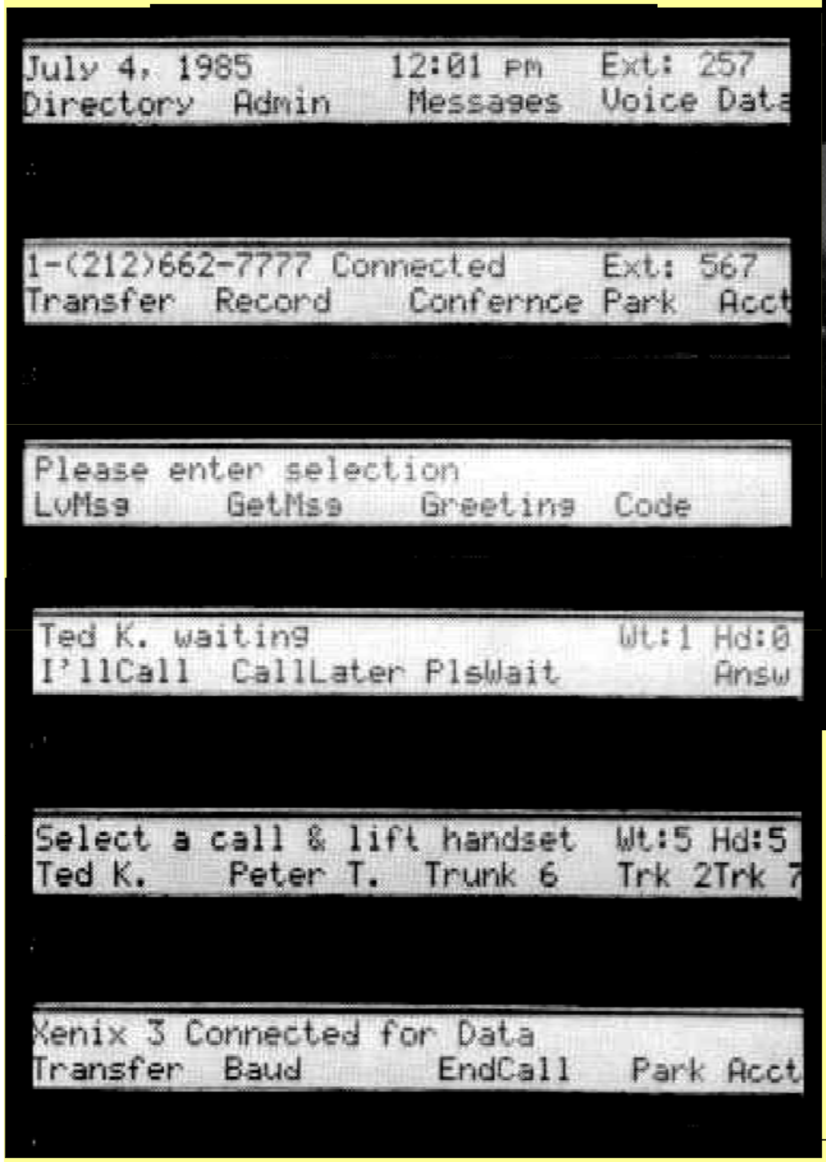
A different approach to automating system testing:

High-volume test 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
LvMsg      GetMsg       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:

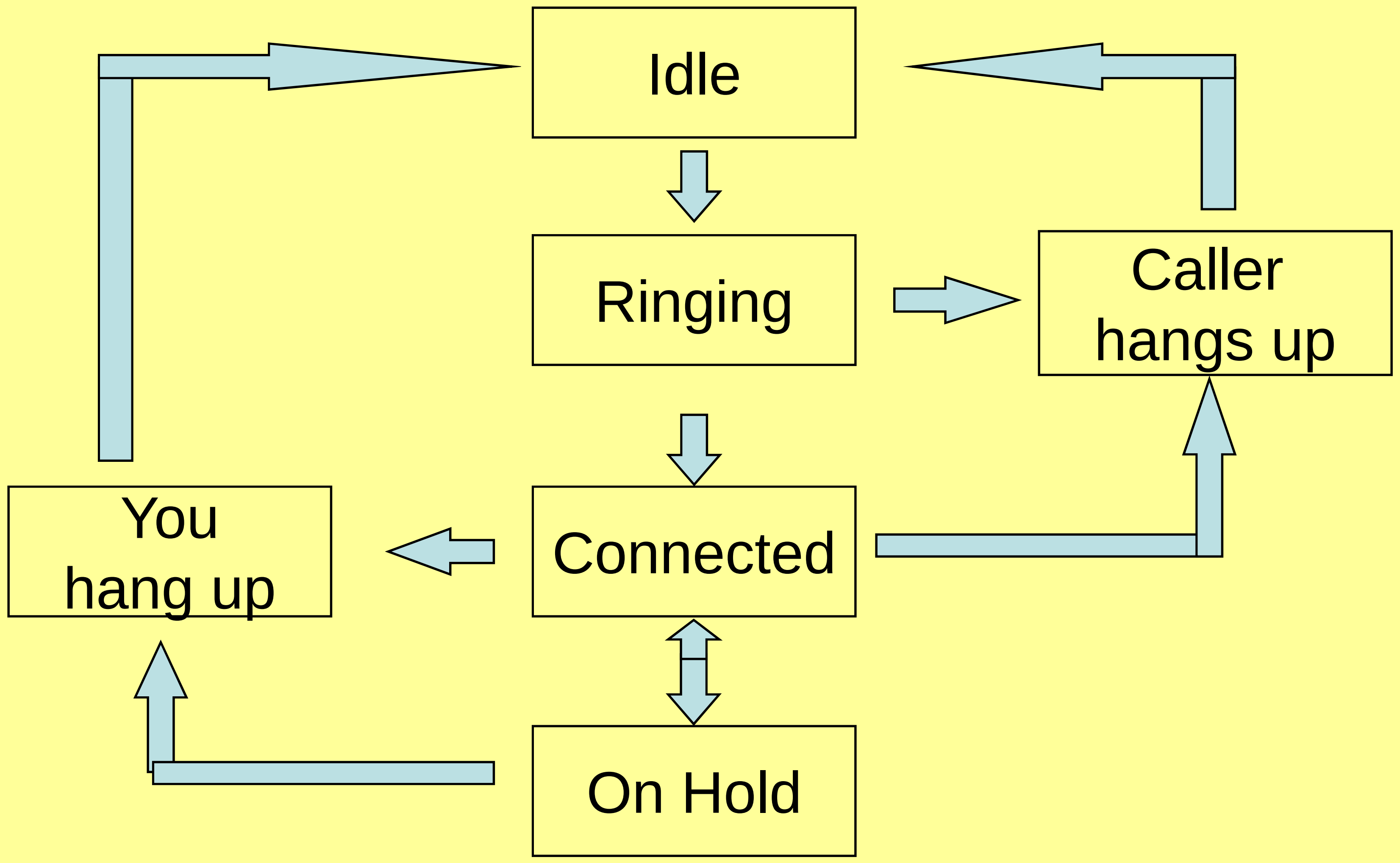
- All individual functions worked
- We had tested all lines and branches.

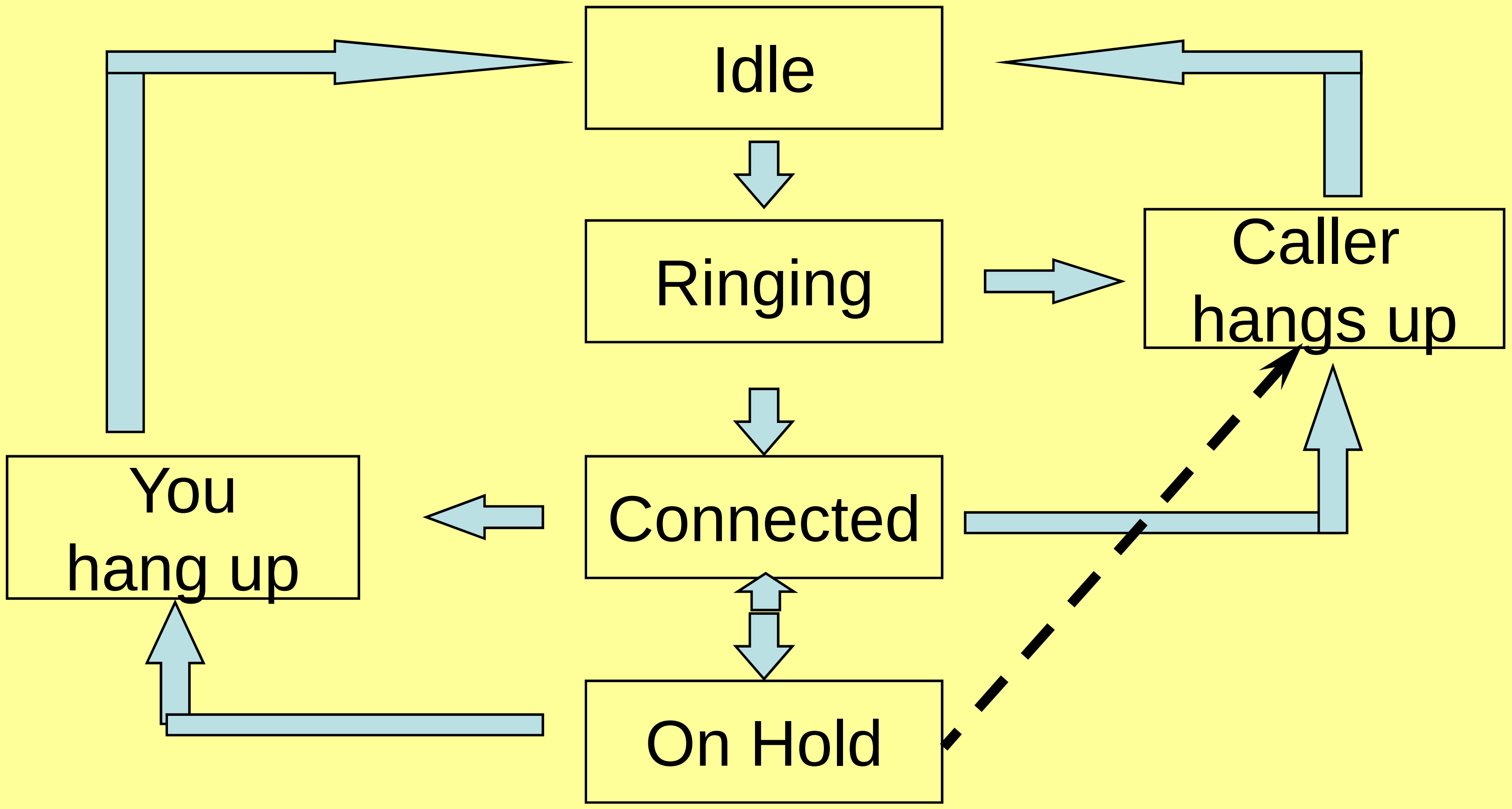
Ultimately, we found the bug in the hold queue

- 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





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 giant 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

(*) By the time this was implemented, I had joined Electronic Arts. This summarizes what colleagues told me, not what I personally witnessed.

Simulator with probes

- 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
- 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

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.
- 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 “Mentsville”, a household-name manufacturer, widely respected for product quality, who chooses to remain anonymous.
- Mentsville 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

A second case study: Long-sequence regression

- 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:
  - these tests are no longer testing for the failures they were designed to expose.
  - these tests add ***nothing*** to typical measures of coverage, because the statements, branches and subpaths within these tests were covered the first time these tests were run in this build.

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.

Long-sequence regression testing

- LSRT has gradually become one of the fundamental techniques relied on by Mentsville
  - gates release from one milestone level to the next.
- Think about testing the firmware in your car, instead of the firmware in Mentsville's devices:
  - fuel injectors, brakes, almost everything is computer controlled
  - for how long a sequence would you want to run LSRT's to have confidence that you could drive the car 5000 miles without failure?
- what if your car's RAM was designed to be reset only after 100,000 miles, not when you turn the car off?

Next challenge:

Tests probably can't be fully specified

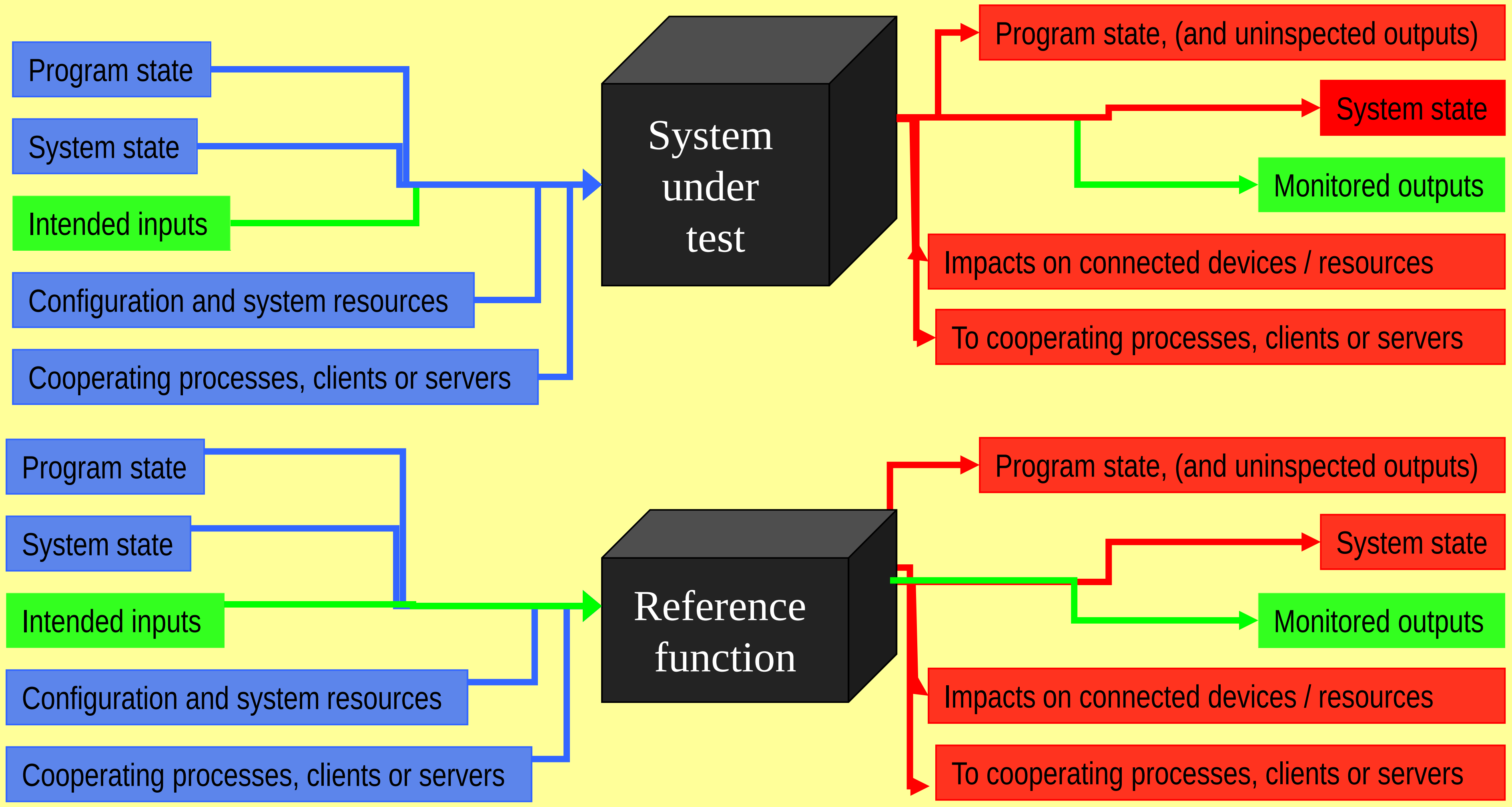
so we probably can't be certain

that a program has actually

passed a test

What do you compare, when you use an oracle?

Based on notes from Doug Hoffman



Program state
System state
Intended inputs
Configuration and system resources
Cooperating processes, clients or servers
System under test
Program state, (and uninspected outputs)
System state
Monitored outputs
Impacts on connected devices / resources
To cooperating processes, clients or servers
Program state
System state
Intended inputs
Configuration and system resources
Cooperating processes, clients or servers
Reference function
Program state, (and uninspected outputs)
System state
Monitored outputs
Impacts on connected devices / resources
To cooperating processes, clients or servers

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 test & reference systems are equivalently configured?

- Does your test documentation specify ALL 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?

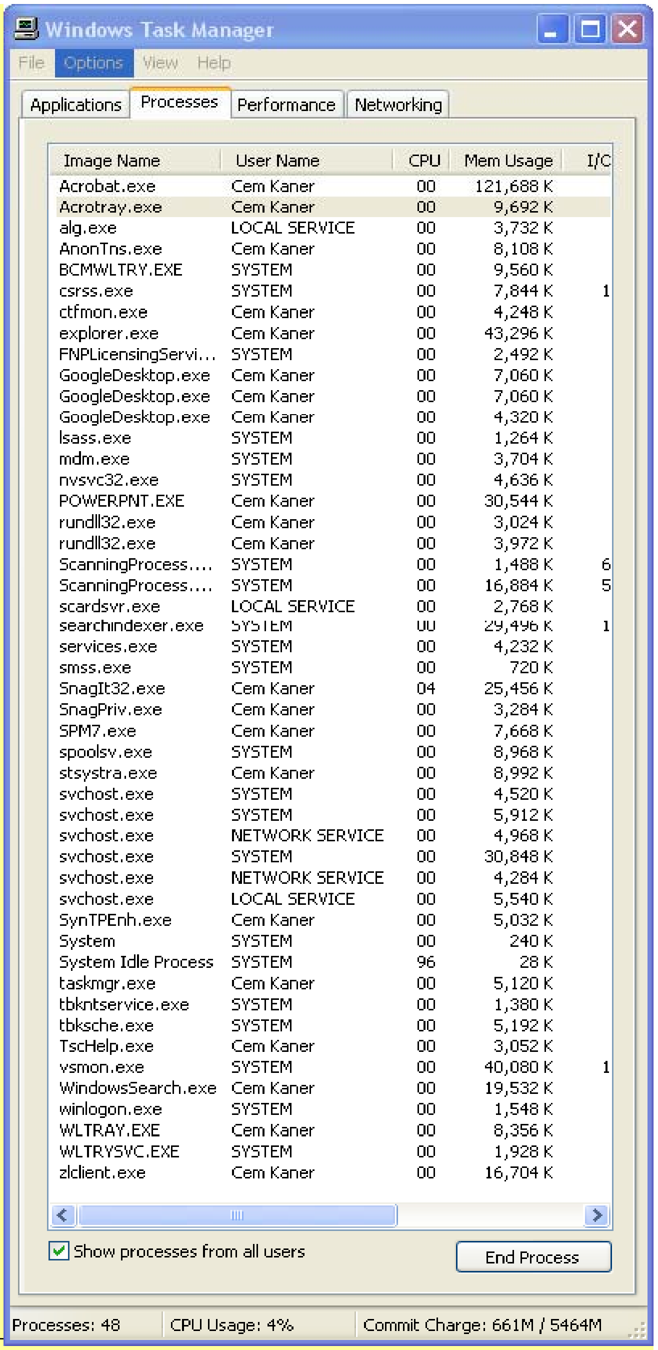

Windows Task Manager
File Options View Help
Applications Processes Performance Networking
Show processes from all users
End Process

Image Name	User Name	CPU	Mem Usage	I/C
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	
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	
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	
nvsvc32.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	
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	
WLTRYSVC.EXE	SYSTEM	00	1,928 K	
zlclient.exe	Cem Kaner	00	16,704 K	

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

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)

- "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.

But....

what about...

regression testing?

(Isn't this the testing

field's bestest practice?

Don't we have to do it?)

Important distinction

- This section is about SYSTEM-LEVEL regression testing
- I agree that UNIT-LEVEL regression testing is a most excellent practice

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

Regression testing

- 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 suitable for regression automation because they provide information that we don't need to collect repeatedly
  - Few tests are worth running on every build.

Cost/benefit the system regression tests

COSTS

- Maintenance of UI / system-level tests is not free
- In practice, we have to do maintenance -- often involving a rewrite of the entire test -- many times.
- We must revise the test, whenever
  - the programmers change the design of the program, even in relatively minor ways.
  - we discover an inconsistency between the program and the test (and the program is correct)
  - we discover the problem is obsolescence of the test
  - we want to integrate new data or new conditions

Cost/benefit the system regression tests

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).

- Process-induced inertia. For example, under our development process, if there is going to be a change, we might have to:
    - rewrite the specification
    - rewrite the related tests (and redocument them)
    - rerun a bunch 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

Cost / benefit of system-level regression

- Perhaps we should concentrate efforts on reducing our UI-level regression testing. Perhaps we should use it for:
  - demonstrations (e.g. for customers, auditors, juries)
  - build verification (that small suite of tests that tells you: *this program is not worth testing further if it can't pass these tests*)
  - retests of areas that seem prone to repeated failure
  - retests of areas that are at risk under commonly executed types of changes (e.g. compatibility tests with new devices)

Cost / benefit of system-level regression

- But not use it for general system-level testing.
- Instead, we could:
  - do 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

Procedural vs risk-focused regression

- 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
- However,
  - The risk-focused tests are different every build and so traditional GUI regression automation is an unavailable strategy

In Summary

- Programmer testing <> system testing
- Recommendations for how to do one of these are probably ineffective and wasteful for the other.
- If we think only about system testing:
  - Validation and accreditation are more important than verification (even though verification may be mandatory)
  - If your work is governed by a contract, teach your lawyer how to specify validation research as part of the testing task
  - High-volume automation can be very useful
  - Try to develop automation that enables you to run tests you couldn't easily / reliably / cost-effectively run before
  - Traditional regression automation is useful for specific tasks but as a general-purpose quality control technique, perhaps it should be reclassified as a worst practice.