Introduction to Exploratory Test Automation
VISTACON, HCMC VIETNAM, September 2010

Cem Kaner, J.D., Ph.D.
Professor of Software Engineering
Florida Institute of Technology

Douglas Hoffman, MSEE, MBA
Fellow of the American Society for Quality
President: Software Quality Methods

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.

An example

- System conversion
  - Database application, 100 transactions, extensively specified (we know the fields involved in each transaction, know their characteristics via data dictionary), 15000 regression tests
  - Should we assess the new system by making it pass the 15000 regression tests?
  - Maybe to start, but what about…
    - Create a test generator to create high volumes of data combinations for each transaction. THEN:
    - Randomize the order of transactions to check for interactions that lead to intermittent failures
  - This lets us learn things we don't know, and ask / answer questions we don't know how to study in other ways

Suppose you decided to never run another regression test. What kind of automation would you do?

Software testing

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

» See appendix for more on basics of test design

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

Copyright © 2010

Exploratory software testing

- is a style of software testing
- that emphasizes the personal freedom and responsibility
- of the individual tester
- to continually optimize the value of her work
- by treating
  - test-related learning,
  - test design,
  - test execution, and
  - test result interpretation
- as mutually supportive activities
- that run in parallel throughout the project.

Exploratory Testing Research Summit, Palm Bay, FL January/February 2006 (http://www.quardev.com/content/whitepapers/exploratory_testing_research_summit.pdf), (http://www.testingreflections.com/node/view/3190) and Workshop on Heuristic & Exploratory Techniques, Melbourne FL, May 2006 (http://www.testingreflections.com/node/view/7386)

ET is an approach to testing, not a technique

- You can use any test technique in an exploratory way or a scripted way
- You can work in an exploratory way at any point in testing

Effective testing requires the application of knowledge and skill

- This is more obvious (but not more necessary) in the exploratory case
- Training someone to be an explorer involves greater emphasis on higher levels of knowledge

What Is Exploratory Test Automation?

- Computer-assisted testing
- That supports learning of new information
- About the quality of the software under test

Typical Testing Tasks

Analyze product & its risks

- benefits & features
- risks in use
- market expectations
- interaction with external S/W
- diversity / stability of platforms
- extent of prior testing
- assess source code

Develop testing strategy

- pick key techniques
- prioritize testing foci

Design tests

- select key test ideas
- create tests for each idea

Run test first time (often by hand)

Evaluate results

- – Troubleshoot failures
- – Report failures

Manage test environment

- – set up test lab
- – select / use hardware/software configurations
- – manage test tools

Keep archival records

- – what tests have we run
- – trace tests back to specs

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

This contrasts the variety of tasks commonly done in testing with the narrow reach of script execution (e.g. UI-level regression automation). This list is illustrative, not exhaustive.

Copyright © 2010

Automating system-level testing tasks

No 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).

Automated system testing doesn't mean automated testing. It means computer-assisted testing

 Copyright © 2010

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.)

 Copyright © 2010

Issues that Drive Design of Test Automation

- **Theory of error**

  What kinds of errors do we hope to expose?
- **Input data**

  How will we select and generate input data and conditions?
- **Sequential dependence**

  Should tests be independent? If not, what info should persist or drive sequence from test N to N+1?
- **Execution**

  How well are test suites run, especially in case of individual test failures?
- **Output data**

  Observe which outputs, and what dimensions of them?
- **Comparison data**

  IF detection is via comparison to oracle data, where do we get the data?
- **Detection**

  What heuristics/rules tell us there **might** be a problem?
- **Evaluation**

  How to **decide** whether X is a problem or not?
- **Troubleshooting support**

  Failure triggers what further data collection?
- **Notification**

  How/when is failure reported?
- **Retention**

  In general, what data do we keep?
- **Maintenance**

  How are tests / suites updated / replaced?
- **Relevant contexts**

  Under what circumstances is this approach relevant/desirable?

Primary driver of our designs

The primary driver of a design is the key factor that motivates us or makes the testing possible.

Our most common primary drivers:

- Theory of error
  - We're hunting a class of bug that we have no better way to find
- Available oracle
  - We have an opportunity to verify or validate a behavior with a tool
- Ability to drive long sequences
  - We can execute a lot of these tests cheaply.

More on ... Theory of Error

Computational errors

Communications problems

- protocol error
- their-fault interoperability failure
- Resource unavailability or corruption, driven by
- history of operations
- competition for the resource
- Race conditions or other time-related or thread-related errors
- Failure caused by toxic data value combinations
- that span a large portion or a small portion of the data space
- that are likely or unlikely to be visible in "obvious" tests based on customer usage or common heuristics

More on … Available Oracle

Typical oracles used in test automation

- Reference program
- Model that predicts results
- Embedded or self-verifying data
- Checks for known constraints
- Diagnostics

» See Appendix for more information on oracles

Additional Considerations

Observation

- What enhances or constrains our ability to view behavior or results?

Troubleshooting support

- Failure triggers what further data collection?

Notification

- How/when is failure reported?

Retention

- In general, what data do we keep?

Maintenance

- How are tests / suites updated / replaced?

Identification of relevant contexts

- Under what circumstances is this approach relevant/desirable?

Some Examples of Exploratory Test Automation

- Disk buffer size
- Simulate events with diagnostic probes
- Database record locking
- Long sequence regression testing
- Function equivalence testing (sample or exhaustive comparison to a reference function)
- Functional testing in the presence of background load
- Hostile data stream testing
- Simulate the hardware system under test (compare to actual system)
- Comparison to self-verifying data
- Comparison to a computational or logical model or some other oracle
- State-transition testing without a state model (dumb monkeys)
- State-transition testing using a state model (terminate on failure rather than after achieving some coverage criterion)
- Random inputs to protocol checkers
  - See Kaner, Bond, McGee, www.kaner.com/pdfs/highvolCSTER.pdf

 Copyright © 2010 Cem Kaner

Disk Buffer Size

- Testing for arbitrary sized buffer writes and reads
- Generate random sized records with random data
- Write records to disk
- Read back records
- Compare written with read data

Simulate Events with Diagnostic Probes

- 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.

Simulate traffic on system, with

- *Settable probabilities of state transitions*
- *Diagnostic reporting whenever a suspicious event detected*

 Copyright © 2010 Cem Kaner

Database Record Locking

Create large random set of records

Launch several threads to

- Select a random record
- Open record exclusive for random time, or
- Open record shared for random time

Long-sequence regression

- 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).
- 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.

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.

Copyright © 2010

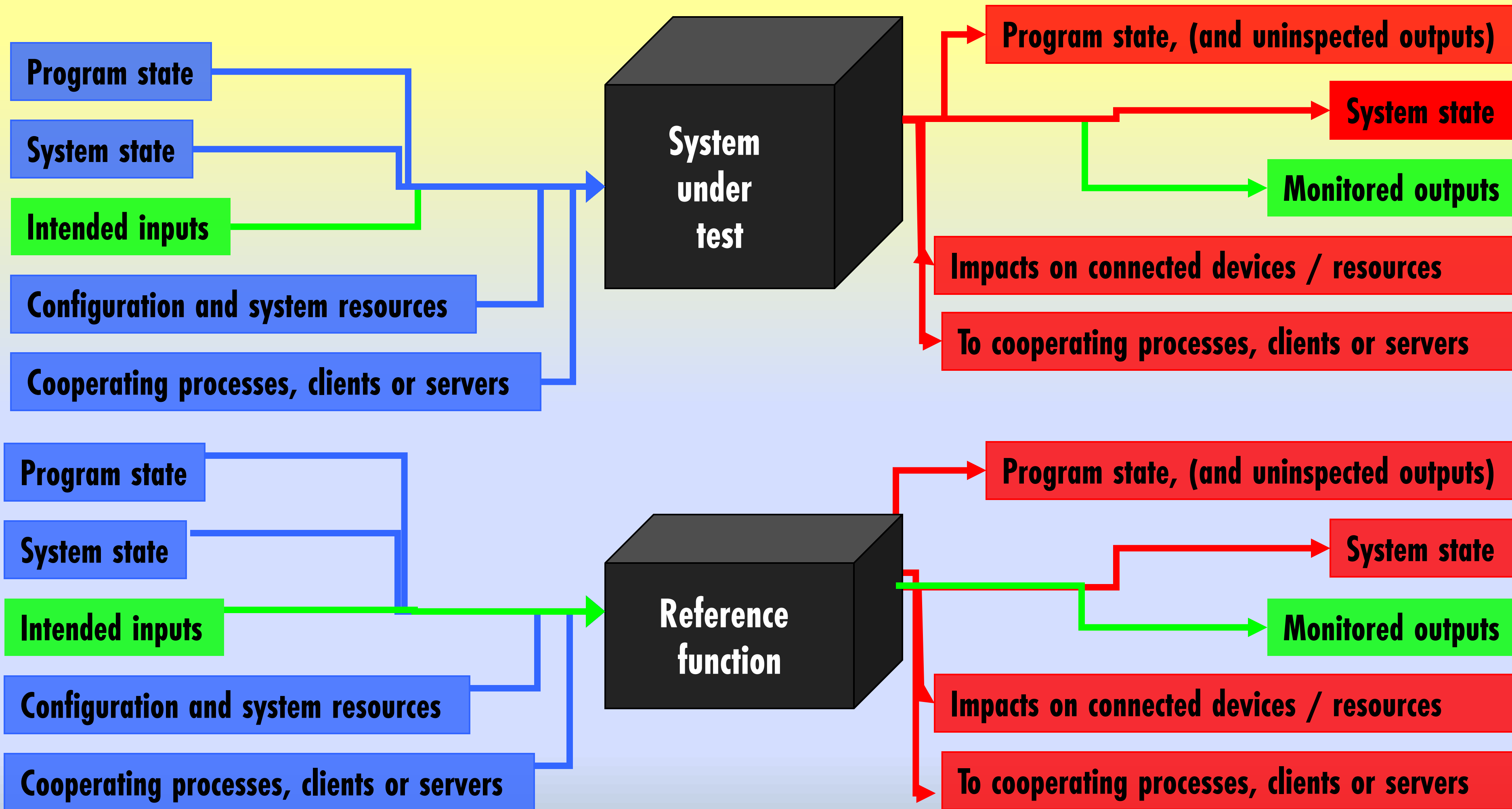
Function Equivalence Testing

- MASPAR (the Massively Parallel computer, 64K parallel processors).
- The MASPAR computer has several built-in mathematical functions. We're going to consider the Integer square root.
- This function takes a 32-bit word as an input. Any bit pattern in that word can be interpreted as an integer whose value is between 0 and 232-1. There are 4,294,967,296 possible inputs to this function.
- Tested against a reference implementation of square root

Function Equivalence Test

- The 32-bit tests took the computer only 6 minutes to run the tests and compare the results to an oracle.
- There were 2 (two) errors, neither of them near any boundary. (The underlying error was that a bit was sometimes mis-set, but in most error cases, there was no effect on the final calculated result.) Without an exhaustive test, these errors probably wouldn't have shown up.
- For 64-bit integer square root, function equivalence tests involved random sample rather than exhaustive testing because the full set would have required 6 minutes x 232 tests.

This tests for equivalence of functions, but it is less exhaustive than it looks



 Copyright © 2010

Functional Testing in the Presence of Background Load

Alberto Savoia ran a series of functional tests

- No failures

Increase background load, replicate the tests

- Initial load increase, no effect
- As load increased significantly, Savoia found an exponential increase in number of functional failures

 Copyright © 2010

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.

Appendix: Basics of Software Testing

From the

Foundations of Software Testing Course

2nd Edition

Software testing

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

Defining Testing

An empirical

- We gain knowledge from the world, not from theory. (We call our experiments, “tests.”)
- We gain knowledge from many sources, including qualitative data from technical support, user experiences, etc.

technical

- We use technical means, including experimentation, logic, mathematics, models, tools (testing-support programs), and tools (measuring instruments, event generators, etc.)

Defining Testing

An empirical, technical …

… investigation

- An organized and thorough search for information.
- This is an active process of inquiry. We ask hard questions (aka run hard test cases) and look carefully at the results.

conducted to provide stakeholders

- Someone who has a vested interest in the success of the testing effort
- Someone who has a vested interest in the success of the product

A law firm suing a company for having shipped defective software has no interest in the success of the product development effort but a big interest in the success of its own testing project (researching the product's defects).

Defining Testing

An empirical, technical investigation conducted to provide stakeholders …

… with information

- The information of interest is **often** about the presence (or absence) of bugs, but other types of information are sometimes more vital to your particular stakeholders
- In information theory, "information" refers to reduction of uncertainty. A test that will almost certainly give an expected result is not expected to (and not designed to) yield much information.

Karl Popper argued that experiments designed to confirm an expected result are of far less scientific value than experiments designed to disprove (refute) the hypothesis that predicts the expectation.

See his enormously influential book, *Conjectures & Refutations.*

Defining Testing

An empirical, technical investigation conducted to provide stakeholders with information …

… about the quality

- Value to some person

of the product or service under test

- The product includes the data, documentation, hardware, whatever the customer gets. If it doesn't all work together, it doesn't work.
- A service (such as custom programming) often includes sub-services (such as support).
- Most software combines product & service

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.

 Copyright © 2010

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

Copyright © 2010

Two examples of test techniques

Scenario testing

Tests are complex stories that capture how the program will be used in real-life situations.

These are combination tests, whose combinations are credible reflections of real use.

These tests are highly credible (stakeholders will believe users will do these things) and so failures are likely to be fixed.

Domain testing

For every variable or combination of variables, consider the set of possible values.

Reduce the set by partitioning into subsets. Pick a few high-risk representatives (e.g. boundary values) of each subset.

These tests are very powerful. They are more likely to trigger failures, but their reliance on extreme values makes some tests less credible.

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

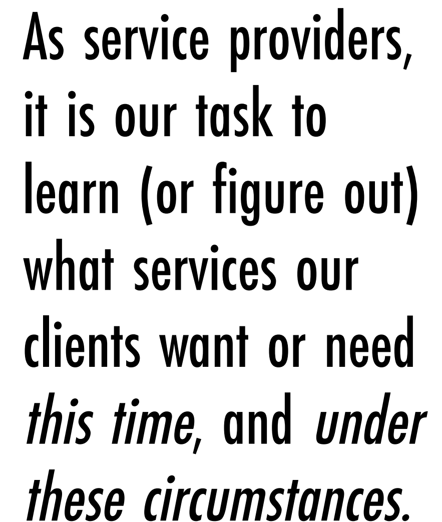
Accountable. You can explain, justify, and prove you ran it

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

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

Testing is always done within a context

- We test in the face of harsh constraints
  - Complete testing is impossible
  - Project schedules and budget are finite
  - Skills of the testing group are limited
- Testing might be done before, during or after a release.
- Improvement of product or process might or might not be an objective of testing.
- We test on behalf of stakeholders
  - Project manager, marketing manager, customer, programmer, competitor, attorney
  - Which stakeholder(s) **this time**?
    - What information are they interested in?
    - What risks do they want to mitigate?



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.

Test design

Think of the design task as applying the strategy to the choosing of specific test techniques and generating test ideas and supporting data, code or procedures:

- Who's going to run these tests? (What are their skills / knowledge)?
- What kinds of potential problems are they looking for?
- How will they recognize suspicious behavior or "clear" failure? (Oracles?)
- What aspects of the software are they testing? (What are they ignoring?)
- How will they recognize that they have done enough of this type of testing?
- How are they going to test? (What are they actually going to do?)
- What tools will they use to create or run or assess these tests? (Do they have to create any of these tools?)
- What is their source of test data? (Why is this a good source? What makes these data suitable?)
- Will they create documentation or data archives to help organize their work or to guide the work of future testers?
- What are the outputs of these activities? (Reports? Logs? Archives? Code?)
- What aspects of the project context will make it hard to do this work?

Appendix: Notes on Oracles

From the

Foundations of Software Testing Course

2nd Edition

Oracles & test automation

We often hear that most (or all) testing should be automated.

- Automated testing depends on our ability to programmatically detect when the software under test fails a test.
- Automate or not, you must still exercise judgment in picking risks to test against and interpreting the results.
- Automated comparison-based testing is subject to false alarms and misses.

Our ability to automate testing is fundamentally constrained by our ability to create and use oracles.

Oracles (Based on notes from Doug Hoffman)

	Description	Advantages	Disadvantages
No Oracle (automated test or incompetent human)	• Doesn't explicitly check results for correctness ("Run till crash")	• Can run any amount of data (limited by the time the SUT takes) • Useful early in testing. We generate tests randomly or from an model and see what happens	• Notices only spectacular failures • Replication of sequence leading to failure may be difficult
No oracle (competent human testing)	• Humans often come to programs with no preset expectations about the results of particular tests. They thus develop ideas about what they are testing and what for, while they are testing.	• See Bolton (2010), "Inputs and expected results", http://www.developsense.com/blog/2010/05/a-transpection-session-inputs-and-expected-results/ • People don't test with "no oracles". They use general expectations and product-specific information that they gather while testing.	• Testers who are too inexperienced, too insecure, or too dogmatic to rely on their wits need more structure.
Complete Oracle	• Authoritative mechanism for determining whether the program passed or failed	• Detects all types of errors • If we have a complete oracle, we can run automated tests and check the results against it	• This is a mythological creature: software equivalent of a unicorn

More types of oracles (Based on notes from Doug Hoffman)

	Description	Advantages	Disadvantages
Heuristic Consistency Oracles	Consistent with • within product • comparable products • history • our image • claims • specifications or regulations • user expectations • purpose	• We can probably force-fit most or all other types of oracles into this structure (classification system for oracles) • James Bach thinks it is really cool • The structure illustrates ideas for test design and persuasive test result reporting	• The structure seems too general for some students (including some experienced practitioners). • Therefore, the next slides illustrate more narrowly-defined examples, inspired by notes from Doug Hoffman
Partial	• Verifies only some aspects of the test output. • All oracles are partial oracles.	• More likely to exist than a Complete Oracle • Much less expensive to create and use	• Can miss systematic errors • Can miss obvious errors

Consistency oracles

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.

All of these are heuristics. They are useful, but they are not always correct and they are not always consistent with each other.

More types of oracles (Based on notes from Doug Hoffman & Michael Bolton)

	Description	Advantages	Disadvantages
Constraints	Checks for • impossible values or • Impossible relationships Examples: • ZIP codes must be 5 or 9 digits • Page size (output format) must not exceed physical page size (printer) • Event 1 must happen before Event 2 • In an order entry system, date/time correlates with order number	• The errors exposed are probably straightforward coding errors that must be fixed • This is useful even though it is insufficient	• Catches some obvious errors but if a value (or relationship between two variables' values) is incorrect but doesn't obviously conflict with the constraint, the error is not detected.
Familiar failure patterns	• The application behaves in a way that reminds us of failures in other programs. • This is probably not sufficient in itself to warrant a bug report, but it is enough to motivate further research.	• Normally we think of oracles describing how the program should behave. (It should be consistent with X.) This works from a different mindset ("this looks like a problem," instead of "this looks like a match.")	• False analogies can be distracting or embarrassing if the tester files a report without adequate troubleshooting.

More types of oracles (Based on notes from Doug Hoffman)

	Description	Advantages	Disadvantages
Regression Test Oracle	• Compare results of tests of this build with results from a previous build. The prior results are the oracle.	• Verification is often a straightforward comparison • Can generate and verify large amounts of data • Excellent selection of tools to support this approach to testing	• Verification fails if the program's design changes (many false alarms). (Some tools reduce false alarms) • Misses bugs that were in previous build or are not exposed by the comparison
Self-Verifying Data	• Embeds correct answer in the test data (such as embedding the correct response in a message comment field or the correct result of a calculation or sort in a database record) • CRC, checksum or digital signature	• Allows extensive post-test analysis • Does not require external oracles • Verification is based on contents of the message or record, not on user interface • Answers are often derived logically and vary little with changes to the user interface • Can generate and verify large amounts of complex data	• Must define answers and generate messages or records to contain them • In protocol testing (testing the creation and sending of messages and how the recipient responds), if the protocol changes we might have to change all the tests • Misses bugs that don't cause mismatching result fields.

Modeling

- A model is a simplified, formal representation of a relationship, process or system. The simplification makes some aspects of the thing modeled clearer, more visible, and easier to work with.
- All tests are based on models, but many of those models are implicit. When the behavior of the program "feels wrong" it is clashing with your internal model of the program and how it should behave).

Characteristics of good models:

- – The representation is simpler than what is modeled: It emphasizes some aspects of what is modeled while hiding other aspects
- – You can work with the representation to make descriptions or predictions about the underlying subject of the model
- – Using the model is easier or more convenient to work with, or more likely to lead to new insights than working with the original.

More types of oracles (Based on notes from Doug Hoffman)

	Description	Advantages	Disadvantages
State Model	• We can represent programs as state machines. At any time, the program is in one state and (given the right inputs) can transition to another state. The test provides input and checks whether the program switched to the correct state	• Good software exists to help test designer build the state model • Excellent software exists to help test designer select a set of tests that drive the program through every state transition	• Maintenance of the state machine (the model) can be very expensive (e.g. the model changes when the program's UI changes.) • Does not (usually) try to drive the program through state transitions considered impossible • Errors that show up in some other way than bad state transition can be invisible to the comparator
Interaction Model	• We know that if the SUT does X, some other part of the system (or other system) should do Y and if the other system does Z, the SUT should do A.	• To the extent that we can automate this, we can test for interactions much more thoroughly than manual tests	• We are looking at a slice of the behavior of the SUT so we will be vulnerable to misses and false alarms • Building the model can take a lot of time. Priority decisions are important.

More types of oracles (Based on notes from Doug Hoffman)

	Description	Advantages	Disadvantages
Business Model	• We understand what is reasonable in this type of business. For example, • We might know how to calculate a tax (or at least that a tax of $1 is implausible if the taxed event or income is $1 million). • We might know inventory relationships. It might be absurd to have 1 box top and 1 million bottoms.	• These oracles are probably expressed as equations or as plausibility-inequalities ("it is ridiculous for A to be more than 1000 times B") that come from subject-matter experts. Software errors that violate these are probably important (perhaps central to the intended benefit of the application) and likely to be seen as important	• There is no completeness criterion for these models. • The subject matter expert might be wrong in the scope of the model (under some conditions, the oracle should not apply and we get a false alarm) • Some models might be only temporarily true
Theoretical (e.g. Physics or Chemical) Model	• We have theoretical knowledge of the proper functioning of some parts of the SUT. For example, we might test the program's calculation of a trajectory against physical laws.	• Theoretically sound evaluation • Comparison failures are likely to be seen as important	• Theoretical models (e.g. physics models) are sometimes only approximately correct for real-world situations

More types of oracles (Based on notes from Doug Hoffman)

	Description	Advantages	Disadvantages
Mathematical Model	• The predicted value can be calculated by virtue of mathematical attributes of the SUT or the test itself. For example: - The test does a calculation and then inverts it. (The square of the square root of X should be X, plus or minus rounding error) - The test inverts and then inverts a matrix - We have a known function, e.g. sine, and can predict points along its path	Good for • mathematical functions • straightforward transformations • invertible operations of any kind	• Available only for invertible operations or computationally predictable results. • To obtain the predictable results, we might have to create a difficult-to-implement reference program.
Statistical	• Checks against probabilistic predictions, such as: - 80% of online customers have historically been from these ZIP codes; what is today's distribution? - X is usually greater than Y - X is positively correlated with Y	• Allows checking of very large data sets • Allows checking of live systems' data • Allows checking after the fact	• False alarms and misses are both likely (Type 1 and Type 2 errors) • Can miss obvious errors

MORE TYPES OF ORACLES (BASED ON NOTES FROM DOUG HOFFMAN)

	Description	Advantages	Disadvantages
Data Set with Known Characteristics	• Rather than testing with live data, create a data set with characteristics that you know thoroughly. Oracles may or may not be explicitly built in (they might be) but you gain predictive power from your knowledge	• The test data exercise the program in the ways you choose (e.g. limits, interdependencies, etc.) and you (if you are the data designer) expect to see outcomes associated with these built-in challenges • The characteristics can be documented for other testers • The data continue to produce interesting results despite many types of program changes	• Known data sets do not themselves provide oracles • Known data sets are often not studied or not understood by subsequent testers (especially if the creator leaves) creating Cargo Cult level testing.
Hand Crafted	• Result is carefully selected by test designer	• Useful for some very complex SUTs • Expected result can be well understood	• Slow, expensive test generation • High maintenance cost • Maybe high test creation cost
Human	• A human decides whether the program is behaving acceptably	• Sometimes this is the only way. “Do you like how this looks?” “Is anything confusing?”	• Slow • Subjective • Credibility varies with the credibility of the human.

Summing up …

- Test oracles can only sometimes provide us with authoritative failures.
- Test oracles cannot tell us whether the program has passed the test, they can only tell us it has not obviously failed.
- Oracles subject us to two possible classes of errors:
  – Miss: The program fails but the oracle doesn't expose it
  – False Alarm: The program did not fail but the oracle signaled a failure

Tests do not provide complete information.

They provide partial information that might be useful.

About Cem Kaner

- Professor of Software Engineering, Florida Tech
- Research Fellow at Satisfice, Inc.

I've worked in all areas of product development (programmer, tester, writer, teacher, user interface designer, software salesperson, organization development consultant, as a manager of user documentation, software testing, and software development, and as an attorney focusing on the law of software quality.)

Senior author of three books:

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

My doctoral research on psychophysics (perceptual measurement) nurtured my interests in human factors (usable computer systems) and measurement theory.