Exploratory Test Automation

CAST August 3, 2010

Douglas Hoffman

Cem Kaner

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

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.

- 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 UI-level regression automation. This list is illustrative, not exhaustive.

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**

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.)

» Harry Robinson's tutorial yesterday provided a lot of thinking along these lines

Primary driver of our designs

- The key factor that motivates us or makes the testing possible.
  - 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

- Reference program
- Model that predicts results
- Embedded or self-verifying data
- Known constraints
- Diagnostics

» For more details: See our Appendix for an excerpt from the new Foundations course! (Brought to you by the letter “B”)

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

1. **Disk buffer size**
2. **Simulate events with diagnostic probes**
3. **Database record locking**
4. **Long sequence regression testing**
5. **Function equivalence testing (sample or exhaustive comparison to a reference function)**
6. **Functional testing in the presence of background load**
7. **Hostile data stream testing**
8. Simulate the hardware system under test (compare to actual system)
9. Comparison to self-verifying data
10. Comparison to a computational or logical model or some other oracle
11. State-transition testing without a state model (dumb monkeys)
12. State-transition testing using a state model (terminate on failure rather than after achieving some coverage criterion)
13. Random inputs to protocol checkers

See Kaner, Bond, McGee, *www.**kaner**.com/pdfs/**high**volCSTER.pdf*

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***

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.

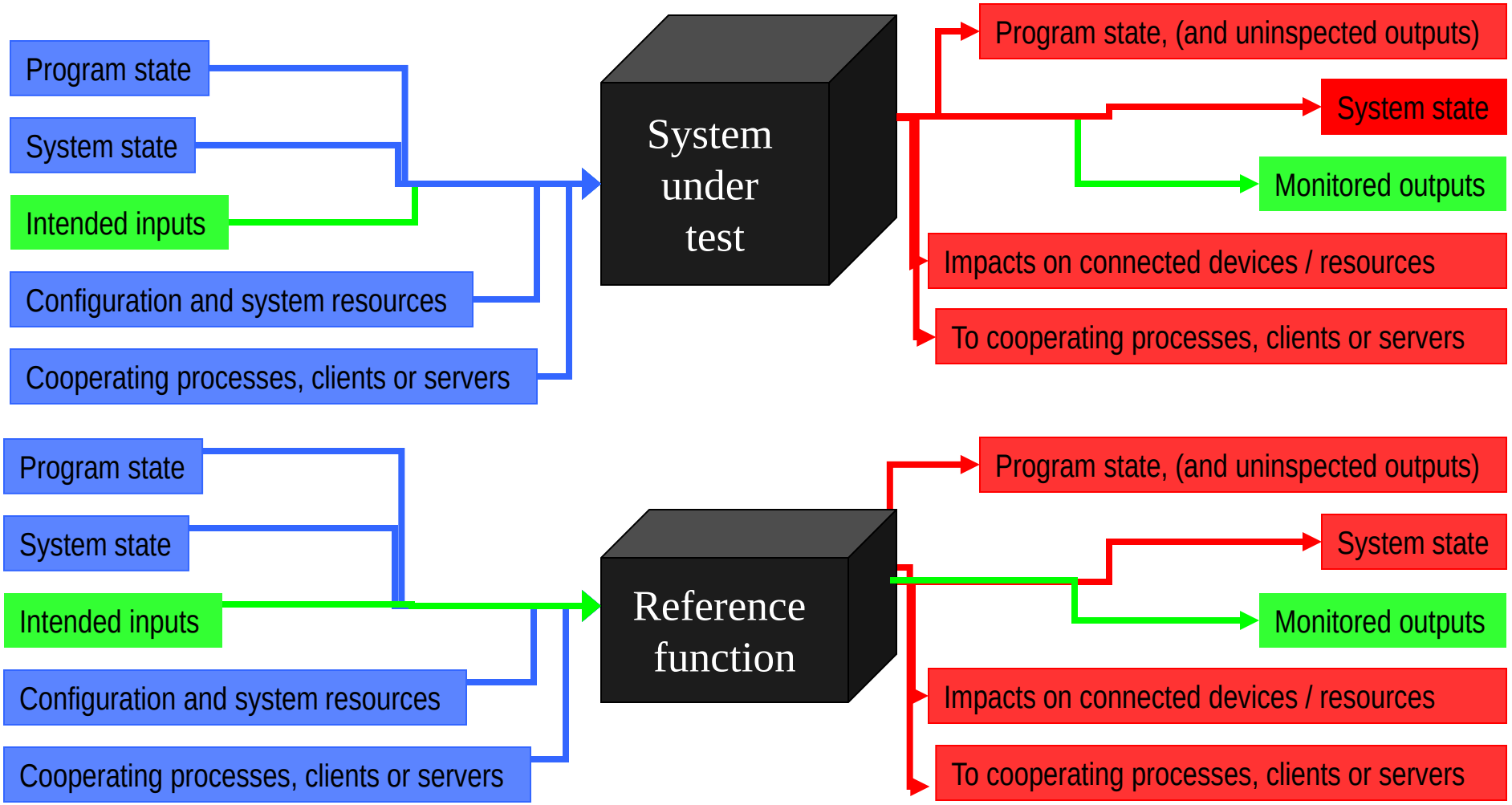
Function Equivalence Testing

- 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 2^{32}-1. There are 4,294,967,296 possible inputs to this function.
- Tested against a reference implementation of square root

Function Equivalence Test

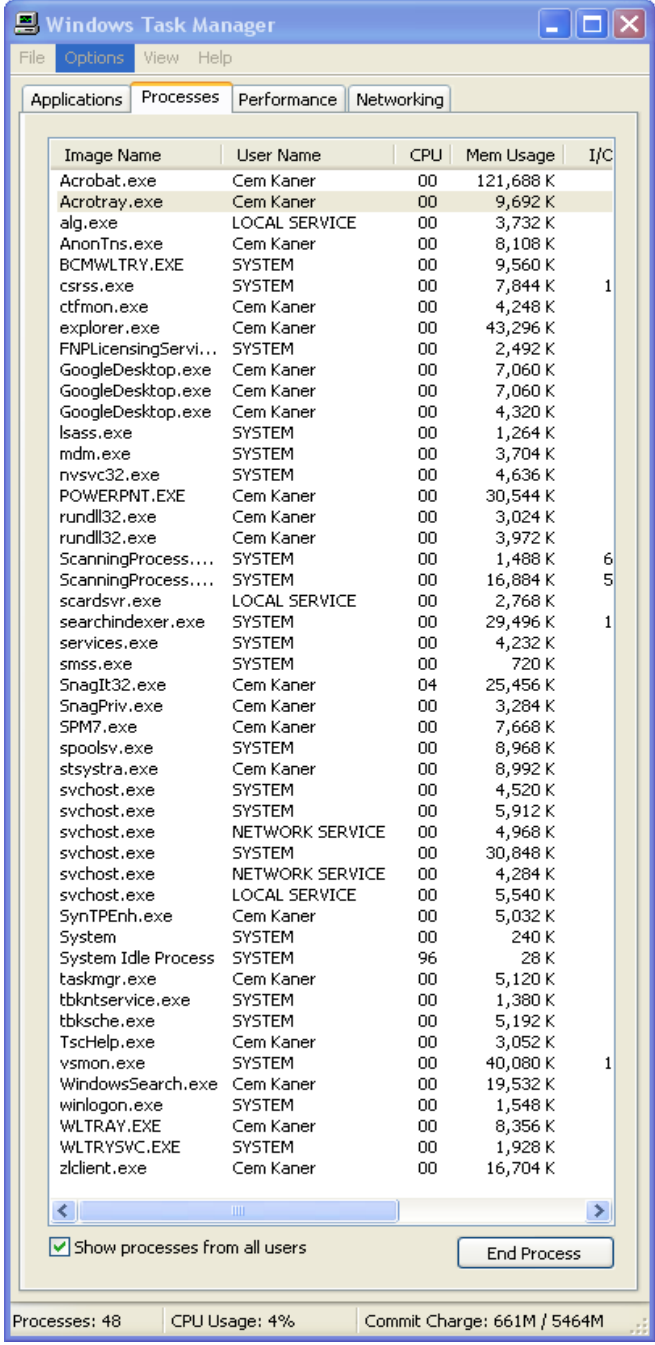
- The 32-bit tests took the computer only 6 minutes to run the tests and compare the results to an oracle.
- There were 2 (two) errors, neither of them near any boundary. (The underlying error was that a bit was sometimes mis-set, but in most error cases, there was no effect on the final calculated result.) 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 2^{32} tests.

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



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?



Functional Testing in the Presence of Background Load

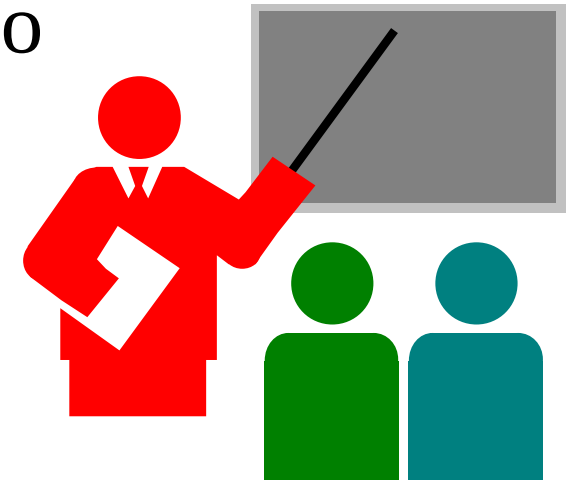
- Alberto Savoia ran a series of functional tests
  - No failures
- Increase background load, replicate the tests
  - Initial load increase, no effect
  - As load increased significantly, Savoia found an exponential increase in number of functional failures

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.

Summary

- Not all automated tests have to do the same thing each time
- Many different ways to explore using automation
  - Looking faster and more deeply
  - Working in areas not humanly accessible



Appendix: More Notes on Oracles

From the

Foundations of Software Testing Course

2nd Edition

Douglas Hoffman & Cem Kaner

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.

You can use all of the oracles in similar ways:

- **Do research** to understand the real-world expectations (what we should expect from this product, this product's competitors, previous versions of this product, etc.)
- **Design tests** to check the match to our expectations – OR—
- **Evaluate the program and then write bug reports** that explain ways in which we are disappointed with the product in terms of mismatch to our expectations (with description of the research basis for those expectations)

Another look at oracles (Based on notes from Doug Hoffman)

	Description	Advantages	Disadvantages
No Oracle	• 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
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
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

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)

	Description	Advantages	Disadvantages
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
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 not obviously wrong, the error is not detected.

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 • Embedded RNG seed (recover original data)	• 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 do not 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).

What might we model in an oracle?

- The physical process being emulated, controlled or analyzed by the software under test
- The business process being emulated, controlled or analyzed by the software under test
- The software being emulated, controlled, communicated with or analyzed by the software under test
- The device(s) this program will interact with
- The reactions or expectations of the stakeholder community
- The uses / usage patterns of the product
- The transactions that this product participates in
- The user interface of the product
- The objects created by this product

What aspects of the things we model might guide our creation of a model?

- Capabilities
- Preferences
  - Competitive analysis
  - Support records
- Focused chronology
  - Achievement of a task or life history of an object or action
- Sequences of actions
  - Such as state diagrams or other sequence diagrams
  - Flow of control
- Flow of information
  - Such as data flow diagrams or protocol diagrams or maps
- Interactions / dependencies
  - Such as combination charts or decision trees
  - Charts of data dependencies
  - Charts of connections of parts of a system
- Collections
  - Such as taxonomies or parallel lists
- Motives
  - Interest analysis: Who is affected how, by what?

What makes these models, *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 if the program UI is changing • 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
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

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

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

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 and need
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 • Not necessarily credible or authoritative

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.