Using Oracles in Test Automation

Douglas Hoffman, BACS, MSEE, MBA
Software Quality Methods, LLC.
24646 Heather Heights Place
Saratoga, California 95070-9710
Phone 408-741-4830
Fax 408-867-4550
doug.hoffman@acm.org
www.SoftwareQualityMethods.com

Douglas Hoffman is an independent consultant with Software Quality Methods, LLC. He has been in the software engineering and quality assurance fields for 30 years and now is a management consultant in strategic and tactical planning for software quality. He is a past Chairman of the Santa Clara Valley Software Quality Association (SSQA), a Task Group of the American Society for Quality (ASQ) and has just completed two years as Chairman of the Silicon Valley Section of ASQ. He has been a presenter and participant at dozens of software quality conferences and has been Program Chairman for several international conferences on software quality. He is a member of the ACM and IEEE and is active in the ASQ as a Senior Member, participating in the Software Division, the Silicon Valley Section, and the Software Quality Task Group. He has earned a BA in Computer Science, an MS in Electrical Engineering, an MBA, a Certificate from ASQ in Software Quality Engineering, and has been a registered ISO 9000 Lead Auditor.

Douglas' experience includes consulting, teaching, managing, and engineering across the computer systems and software industries. He has twenty years experience in creating and transforming software quality and development groups, and has worked as an independent consultant for the last ten years . His work in corporate, quality assurance, development, manufacturing, and support organizations provides a broad technical and managerial perspective on the computer industry.

Key points to get from this paper:

- Automated tests should be planned and engineered
- Most automated tests need verification steps to be useful
- More powerful automated tests are made possible with oracles
- There are different types of oracles we can use
- There are several strategies for verification in automated tests

Using Oracles in Test Automation

PNSQC 2001

Douglas Hoffman

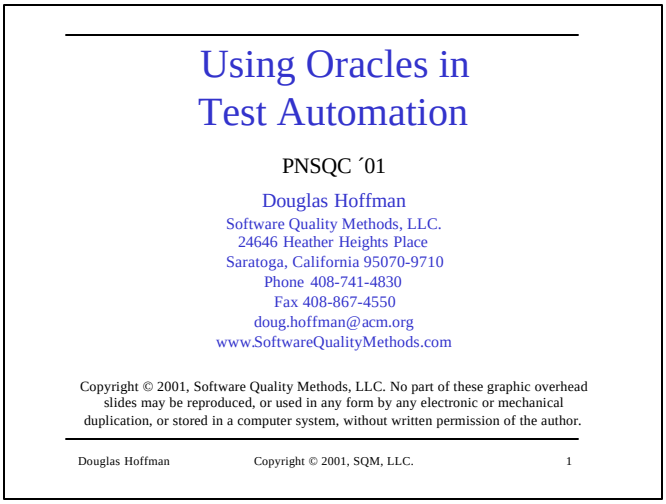
Summary

Software test automation is often a difficult and complex process. The most familiar aspects of test automation are organizing and running of test cases and capturing and verifying test results. When we design a test we identify what needs to be verified. A set of expected result values are needed for each test in order to check the actual results. Generation of expected results is often done using a mechanism called a test oracle. This paper describes the purpose and use of oracles in automated software verification and validation. Several relevant characteristics of oracles are included with the advantages, disadvantages, and implications for test automation.

Real world oracles vary widely in their characteristics. Although the mechanics of specific oracles may be vastly different, a few classes can be identified which correspond with automated test strategies. Oracles are categorized based upon the strategy for verification using the oracle. Thus, a verification strategy using a lookup table to compare expected and actual results can use the same type of oracle as one that uses an alternate algorithm implementation to compute them.

Background

Most software test automation begins with conversion of existing (manual) tests. In many instances the expected results are either embedded in the test or captured in a logging file for later verification. This approach is straightforward and can be used for a variety of tests. Tests automated in this way run automatically, but they are less likely to find errors than their manual counterpart for several reasons. A test automated this way will do the same thing each time it runs, especially since the inputs are provided by an automaton. The test will also verify only and exactly the results that we write into the test. Automated result comparison depends on having the results in a computer readable form.



Using Oracles in Test Automation

PNSQC ´01

Douglas Hoffman
Software Quality Methods, LLC.
24646 Heather Heights Place
Saratoga, California 95070-9710
Phone 408-741-4830
Fax 408-867-4550
doug.hoffman@acm.org
www.SoftwareQualityMethods.com

Copyright © 2001, Software Quality Methods, LLC. No part of these graphic overhead slides may be reproduced, or used in any form by any electronic or mechanical duplication, or stored in a computer system, without written permission of the author.

Douglas Hoffman Copyright © 2001, SQM, LLC. 1

For manual tests, a person provides the input and evaluates results, while automated tests use programs to do the work. A person will not do exactly the same thing the same way even when they try, while an automaton will tend to do exactly the same thing every time. Testers mis-key and correct their typing and people are also subject to variations in timing, so some possibly material characteristics can change simply because we aren't automatons. A person running manual tests can easily vary the test exercise and evaluate the responses of the software under test (SUT). Manually rerunning tests introduces new variations and exercises, improving the likelihood of finding new problems even with an old test.

A person running a manual test is also able to perceive unexpected behaviors for which an automated verification doesn't check. This is a powerful advantage for manual tests; a person may notice a flicker on the screen, an overly long pause before a program continues, a change in the pattern of clicks in a disk drive, or any of dozens of other clues that an automated test would miss. The author has seen automated tests "pass" and then crash the system, a device, or the SUT immediately afterwards. Although not every person might notice these things and any one person might miss them sometimes, an automated test only verifies those things it was originally programmed to check. If an automated test isn't written to check timing, it can never report a time delay.

We need to approach automated testing differently from manual testing if we want to get equal or better tests.

What Makes Automated Tests Different

In order to check the results, inputs to the SUT must be tracked and some means of generating a prediction of the resultant behaviors provided for some or all of the same dimensions. In a manual test, the tester usually controls or checks preconditions and inputs, and can quickly adjust the system when they encounter unexpected results. An automated test must rely upon the test design and system setup to control the important inputs. It must also include some mechanism for knowing or getting the expected results (typically from an oracle). Regardless of the test exercise, an automated test will be poor with poorly selected inputs or results, with poor results oracles, or if there is limited visibility into the relevant values.

Automated verification assumes that we know what to check to know whether the software did what it was supposed to do. It also assumes that we know the correct values for whatever we check. We identify what the software was supposed to change (or not change) when we design the test. Then we check it to confirm that it happened after we run the test. However, when the SUT fails it can change almost anything. Our tests should look for these failures because it does little good for a test to encounter an error and then ignore it. When manual testing, a person can be effective without knowing in advance exactly what the test results are. The tester will learn and adapt, checking different things at different times. If something doesn't seem right, the tester can dig into the details to figure out if the SUT behaved correctly. This is not true for automated tests. A tester can manually verify one or several conditions and data values, and doesn't have to do the same thing each time a test is run. For automated tests, this is established in advance.

The question of what to check is a big one that's often overlooked. We may have successfully entered an order and properly updated inventory counts and financial books, but, did we check to

see if there was any effect on other orders? We successfully added a user and set their permissions, but, did the permissions change for any other users? Software errors can cause an infinite variety of changes in the system and we can check only a few. With automated tests we define what to check in advance and limit the verification to exactly that.

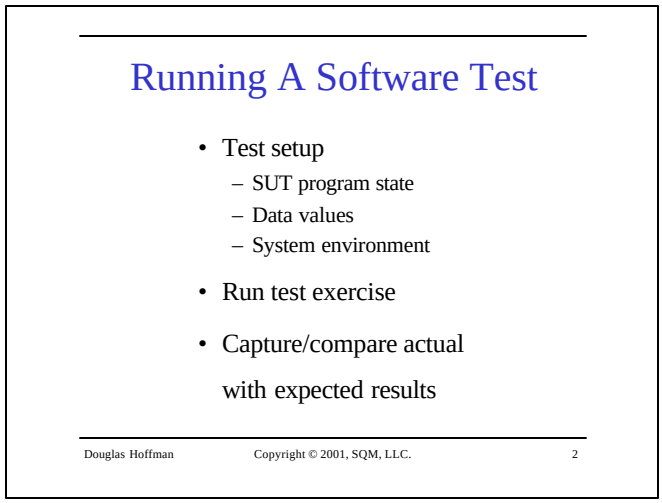
One reason we avoid checking many possible results is the difficulty of knowing what the results should be. This is the role of an oracle – to generate the expected result (or at least answer whether the actual result seems plausible). An oracle is critically important if we are to create automated tests that are equal or better than manual tests. We can generate millions of inputs in an automated test and verify proper SUT behavior using an oracle. Automated tests may be able to recover from unexpected responses if we have oracles that predict correct responses (and thus can show us a path to get back on track). Usually we need multiple oracles for these more sophisticated tests because we verify more than one thing.

Automated verification of results can also be quite difficult technically. How do you verify that the sound track synchronized with the motion picture? How do you verify that the image printed on the page is what you expected? Testers' perceptions can easily and effectively analyze results that we have a tremendous amount of difficulty getting a computer to verify. Some test automation problems are not cost effective to solve today.

This is not to say that automated tests aren't useful. Automated tests can be very powerful for finding certain kinds of errors. Manually rerunning the same tests every time anything changes is time consuming and boring for people, but machines are designed for doing this kind of task. People are also easily trained about what to expect from a test and can cognitively miss seeing errors after only a few repetitions. Machines do what we tell them to do, as many times as we want. And for massive numbers of data points, test iterations, and combinations there may not be any way to run tests except using automation. Some testing problems simply cannot be solved manually, such as performance analysis and system load testing.

Software Test Models

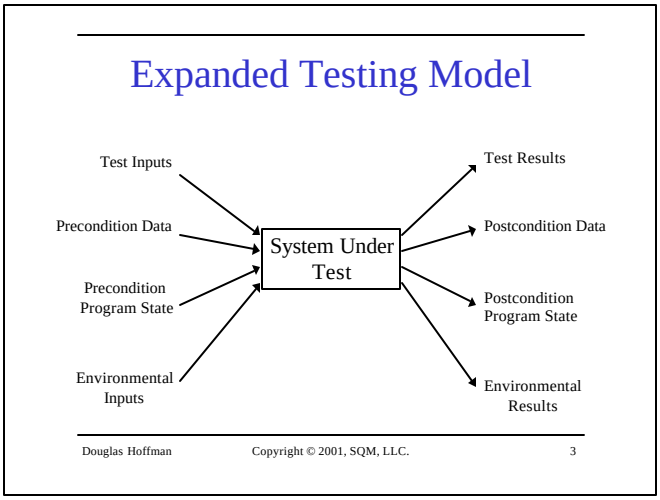
A software test consists of three steps: setting up the conditions in the system, providing stimulation to the SUT, and observing the results. This applies for manual and automated tests (shown in **Slide 2**). The setup creates the conditions necessary for some errors to manifest (assuming the errors are there). The test run exercise takes the SUT through the suspect code. Then we can confirm whether there are errors in the software.



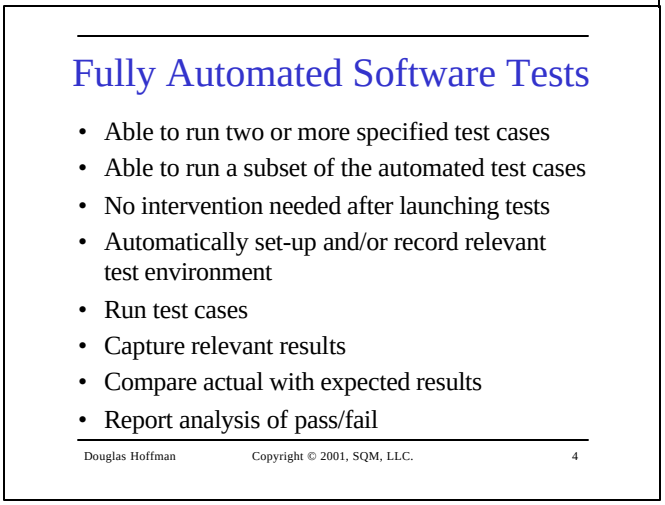
The test setup is often neglected, assuming that a test will work from whatever state the SUT, data, and system are in. Yet, it is obvious that software will do things differently based on

the starting state of the SUT and system. Is the SUT running? Is the account we are accessing already in the database, or do we need to add it? Is an error dialog currently on the screen? What permissions does the current user have? Are all the necessary files on the system? What is the current directory? Testers note and correct for all such conditions when manually testing. But these are potentially major issues for automated tests.

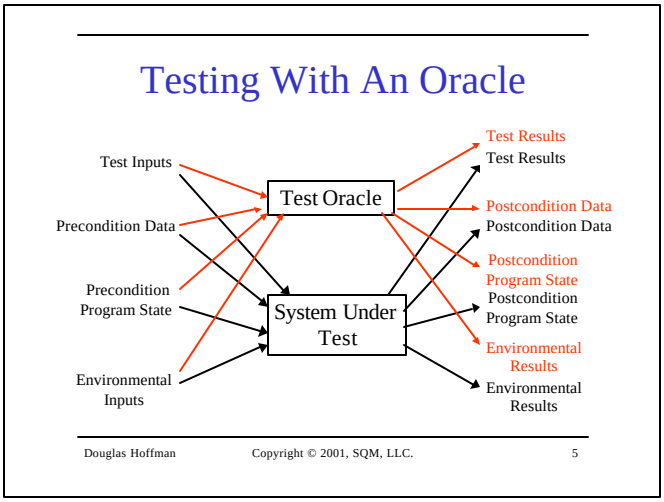
Although the test designer typically is conscious only of the values directly given to the SUT, the SUT behavior is influenced by its data, program state, and the configuration of the system environment it runs in. Test setup consists of monitoring or controlling things in all these domains. The domains include the data and program state information, which are somewhat manageable by the software test automation management system and within the automated tests. The environment, however, gets very difficult to scope out and manage. The SUT may behave differently based on the computer system environment; operating system version, size of memory, speed of the processors, network traffic, what other software is installed, etc. Even more difficult to analyze or manage is the physical environment – I've worked on errors due to temperature, magnetic fields, electrostatic discharge, poor quality electric grounding, and other physical environmental factors that caused software errors. These errors may be unusual, but they occur, so we need to keep them in mind when automating tests. **Slide 3** illustrates a model of actual inputs and results in a software test.



The term "automated software test" has many different meanings, depending upon the speaker and context. For the purposes of this paper, automated software testing has the eight characteristics shown in ***Slide 4****. The test consists of performing some exercise of the SUT, observing some results, comparing them with expected result values, and reporting the outcome.*



The running of an automated test exercise is often the easiest part of testing. Given that the SUT, data, and system are in the proper initial state, the automated test can feed the test data into the SUT[1]. (For the purposes of this paper, I'm going to assume that it is that simple[2].) The exercise puts the SUT through its paces and either encounters errors or not.



Some of the biggest difficulties in software test automation are in knowing what results are expected from the SUT. There are many issues with the huge number of potentially relevant results and how to record them. As just described, we are dealing with multiple input domains, and not surprisingly, the same domains can be effected by the SUT. (This is especially true when we consider that there may be errors in the SUT, so the actual result is outside of the expected realm.) Often, it is extremely difficult to predict what the SUT should do and what outcomes are expected, even for the set of expected results. **Slide 5** illustrates a model incorporating an oracle to predict expected results from actual inputs.

Several observations can be made from the model. Different types of oracles are needed for different types of software and environments. The domain, range, and form of input and output data varies substantially between programs. Most software has multiple forms of inputs and results so several oracles may be needed for a single software program. For example, a program's direct results may include computed functions, screen navigations, and asynchronous event handling. Several oracles may need to work together to model the interactions of common input values. If we consider a word processor, pagination changes are based upon the data being displayed and characteristics such as the page width, point size, page layout, and font. In Windows, the current printer driver also affects the pagination even when nothing is printed. Just changing the selected printer to one with a different driver can change the pagination of a Word document. Although an oracle may be excellent at predicting certain results, only the SUT running in the target environment will process all of the inputs and provide all of the results.

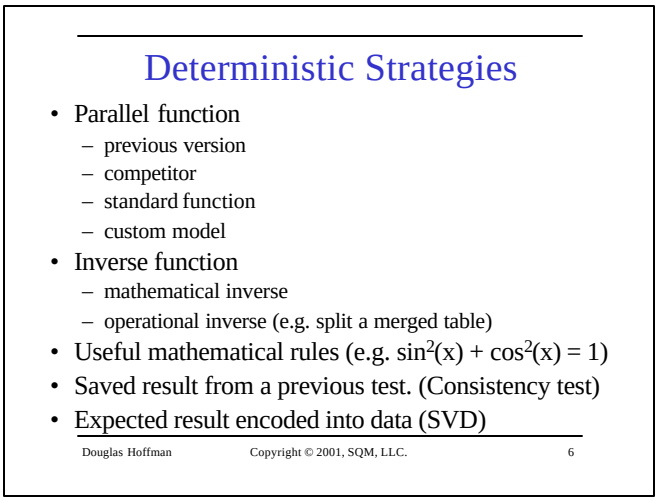
Characteristics of Oracles

When we think of software testing and the test results, it's usually in a binary sense: the result is right or wrong; the test passed or failed. We generally don't allow for "maybe it's OK" or "possibly right" as outcomes. We consider the test outcomes to be deterministic; there is one right answer [and we know what it is]. **Slide 6** lists some examples of such deterministic verification strategies. Each example provides some means for determining whether or not a test

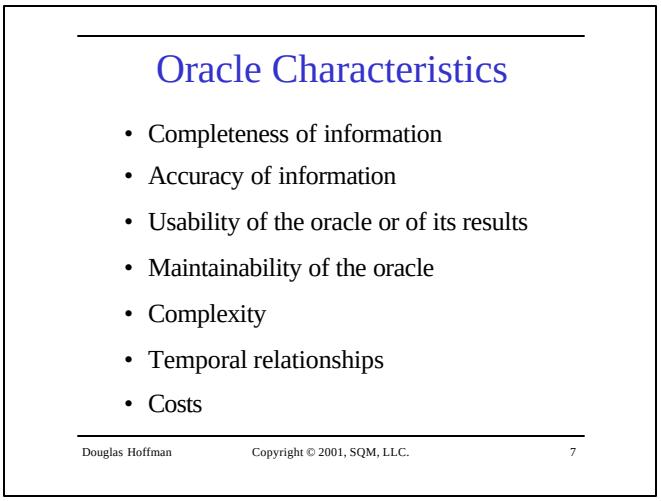
[1] "data" in this case includes input data, control information, and whatever else is needed for the test to stimulate the SUT.

[2] Much of the work in test automation to date has focused on the mechanics of feeding and manipulating data and controls in the test exercise. The focus here is on oracles and verification of the test results.

result is correct. It is useful to note that although the strategy allows us to pass or fail a particular result, in many cases there are ways that the SUT can give us a wrong result, and yet the test can pass (e.g., an undetected error in the previous version or the competitor's product).



There are several interesting characteristics relating an oracle to the SUT. **Slide 7** provides a list of some useful characteristics based on the correspondence between the oracle and the SUT. The results predicted by an oracle can range from having almost no relationship to exact duplication of the SUT behaviors. Completeness, for example, can range from no predictions (which may not be very useful) to exact duplication in all results categories (an expensive reimplementation of the SUT).



Completeness of information:

- Input Coverage
- Result Coverage
- Function Coverage
- Sufficiency
- Types of errors possible

Accuracy of information:

- How similar to SUT
  - Arithmetic accuracy
  - Statistically similar
- How independent from SUT
  - Algorithms
  - Sub-programs & libraries
  - System platform
  - Operating environment
  - Close correspondence makes common mode faults more likely and reduces maintainability
- How extensive
  - The more ways in which the oracle matches the SUT, i.e. the more complex the oracle, the more errors
- Types of possible errors
  - Misses actual wrong value
  - Flags correct data as an error
  - Some oracles may allow one or both types of errors

Usability of the oracle or of its results:

- Form of information
  - Bits and bytes
  - Electronic signals
  - Hardcopy and display
- Location of information
- Data set size
- Fitness for intended use
- Availability of comparators
- Support in SUT environments

Maintainability of the oracle:

- COTS or custom
  - Custom oracle can become more complex than the SUT
  - More complex oracles make more errors
- Cost to keep correspondence through SUT changes
  - Test exercises
  - Test data
  - Tools
- Ancillary support activities required

Complexity:

- Correspondence with SUT
- Coverage of SUT domains and functions
- Accuracy of generated results
- Maintenance cost to keep correspondence through SUT changes
  - Test exercises
  - Test data
  - Tools
- Ancillary support activities required

Temporal relationships:

- How fast to generate results
- How fast to compare
- When is the oracle run
- When are results compared

Costs:

- Creation or acquisition costs
- Maintenance of oracle and comparitors
- Execution cost
- Cost of comparisons
- Additional analysis of errors
- Cost of misses
- Cost of false alarms

The more complete and accurate an oracle is, the more complex it has to be. Indeed, if the oracle exactly predicts all results from the SUT it will be at least as complex. In some cases an oracle is more complex than the SUT when the simulators, operating systems, etc., are all considered. The better that an oracle provides expected results, the more complex it is and the more likely that detected differences are due to errors in the oracle rather than the SUT. Likewise, the more an oracle predicts about program state and environment conditions, the more sensitive the oracle is to changes in the SUT and operating environment. This dependence makes the oracle more complex and more difficult to maintain. It also means that errors may be missed because of common mode errors where both the SUT and the oracle generate the same wrong result due to sharing of a component with an error.

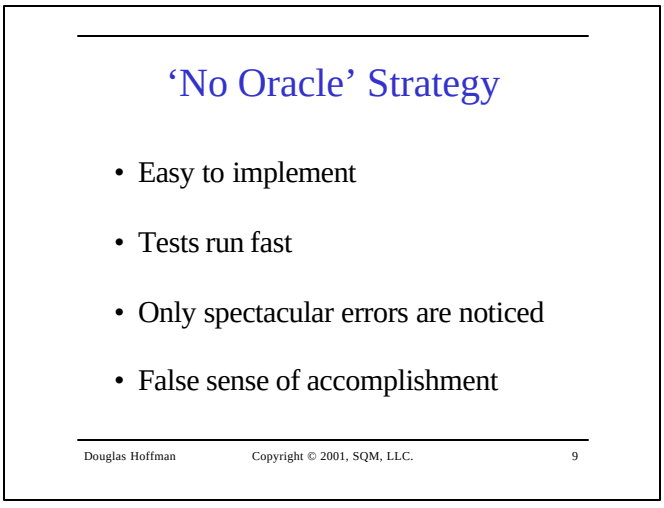
Oracle Strategies For Automated Test Verification

Four types of oracle strategies (and not using any oracle) are identified and outlined in **Table 1**. The strategies are labeled True, Consistency, Self Referential, and Heuristic. Each strategy is expanded upon below.

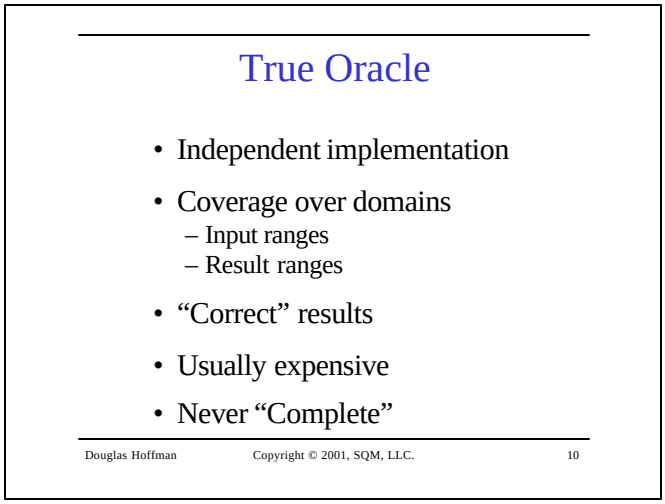
	No Oracle	True Oracle	Consistency	Self Referential (SVD)	Heuristic
Definition	• Doesn't check correctness of results	• Independent generation of all expected results	• Verifies current run results with a previous run (Regression Test)	• Embeds answer within data in the messages	• Verifies some characteristics of values
Advantages	• Can run any amount of data (limited only by the time the SUT takes)	• All encountered errors are detected	• Fastest method using an oracle • Verification is straightforward • - Can generate and verify large amounts of data	• Allows extensive post-test analysis • Verification is based on message contents • Can generate and verify large amounts of complex data	• Faster and easier than True Oracle • Often much less expensive to create and use
Disadvantages	• Only spectacular failures are noticed.	• Expensive to implement • Complex and often time-consuming when run	• Original run may include undetected errors	• Must define answers and generate messages to contain them	• Can miss errors • Can miss systematic errors

Table 1: Five Oracle Strategies

It is possible to automate the running of a test without checking results. I have encountered organizations that knew and planned such automated tests, and a few that simply hadn't thought to verify test results from their automation. This approach gets around the problems of false error reports and the costs of maintaining the oracle. It also has the advantage that it's easy, inexpensive, and tests run quickly. The major disadvantage is that only a few spectacular errors can be found this way. The automation may give



observers the impression that there is more value to the testing than there really is. There are occasions when the goal is to provide some exercise in the SUT, and the test outcomes are not important (e.g., the tests are needed just to provide a background load). However, effort and activity are not the same as accomplishment. (Running a useless exercise 1,000,000 times each night is still a useless exercise.)



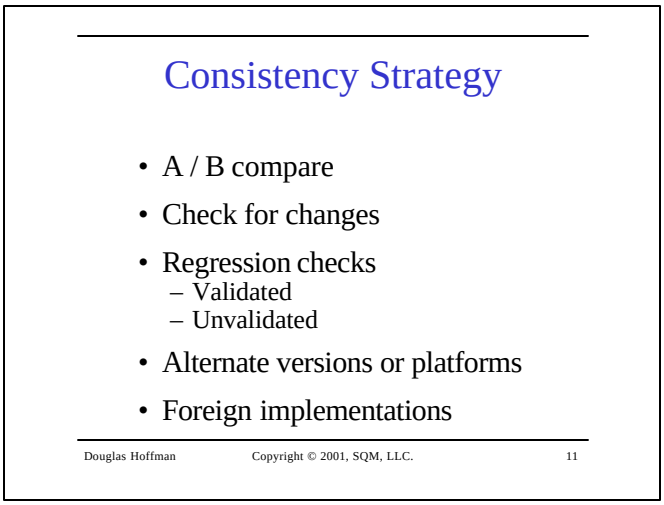
A true oracle faithfully reproduces all relevant results for a SUT using independent platform, algorithms, processes, compilers, code, etc. The same values are fed to the SUT and the oracle for verification. This type of oracle is well suited for verification of an algorithm or subroutine. For a given test case, all values input to the SUT are verified to be correct using the oracle's separate algorithm. The less the SUT has in common with the oracle, the more confidence in the correctness of the results (since common hardware, compilers, operating systems, algorithms, etc., may inject errors that effect both the SUT and oracle the same way). Automated test cases employing true oracles are usually limited by available machine time and system resources, not the oracle itself.

A true oracle may be slow or expensive to use, so tests may use a small sample to limit the amount of test data. One way this is done is for random selection of inputs within ranges where the oracle works. Different inputs can be generated each time the test is run by using a pseudo-random numbers (repeatable sequences of random numbers) to select the input values. Another small sample approach is done by creating a table of inputs with corresponding results from the oracle. Input values chosen from the table are fed to the SUT and the results verified from the table.

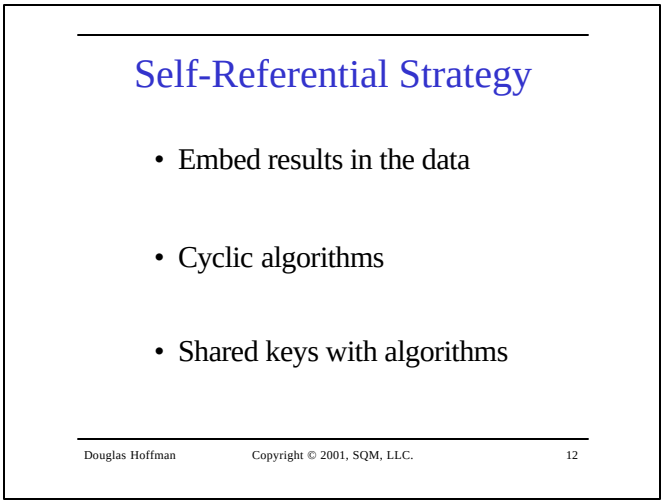
Note that true oracles do not have to be complete to be useful. In fact, a completely replicated system will not provide identical responses over all of the input and result domains described in **Slide 5**. An oracle that works for a subset of input values or covers only one characteristic result can be very effective. A test may cover only a small range of input values or we may check only that the correct sequence of screens appears without need for a complete oracle. Being a true oracle, however, means that for the subset of input values or the sequence of screens we test, the oracle correctly provides the expected results.

The consistency approach uses the results from one test run as the oracle for subsequent tests. Thus, we learn whether the results are the same as before, with differences likely the result of errors. This is probably the most used strategy for automated regression tests, as it is particularly useful for evaluating the effects of changes from one revision to another. Although it doesn't tell us whether the results are actually correct, it does expose differences or changes. The oracle can be a simulator, equivalent product, software from an alternate platform, or an early version of the

SUT. Because we don't need to know if the results are correct, we can use pseudo-random numbers to generate huge volumes of results to be compared. At the same time, the values being compared can include intermediate results, call trees, raw internal data values, or any other data extracted from the SUT. Comparing results between the SUT and the oracle tells us of any changes, which indicates something was fixed or broken. Although historic faults may remain when this technique is used, new faults and side-effects are often exposed and fixes are confirmed.



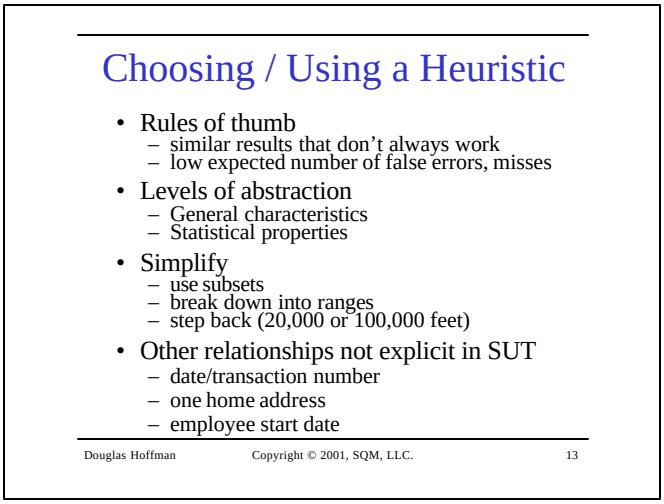
A self-referential strategy builds the expected results (answers) into the data as part of the test mechanism. For example, when testing data communications, we might include the expected communications protocol information in the message we send, so a receiver could confirm that the envelope data matches. In a test for a data base engine, a data field could describe the data base linkages expected between fields or records. The random number "seed" could be included in the randomly generated data set so that the random number series can be rerun. In this strategy, the tests are designed so that they create records with specific characteristics, and those characteristics are included within the records themselves. We then use pseudo-random numbers to generate arbitrary populations of test data. An independent analysis can be run to identify problems or inconsistencies.



A heuristic is a general "rule of thumb" we can use to quickly assess whether the result is likely to be correct or incorrect. It is not guaranteed to find all errors and it may flag correct results as errors. This strategy verifies results using simpler algorithms or consistency checks based on a heuristic. For example, a heuristic strategy for a USA zip code might check that the values have five or nine digits. A heuristic for a *sin()* function could use the mathematical identity that $sin^2(x) + cos^2(x) = 1$ to verify it (assuming that the implementation of *sin()* and *cos()* are not based on that relationship). Although the heuristic approach may accept results that are incorrect or reject results that are correct, the oracle is easy to implement (especially when compared to a true oracle), runs much faster, and can be used to quickly find many classes of errors.

A heuristic may be based on incidental relationships associated with the data. For example, if transaction numbers are generated sequentially when the transactions begin, then sorting by date

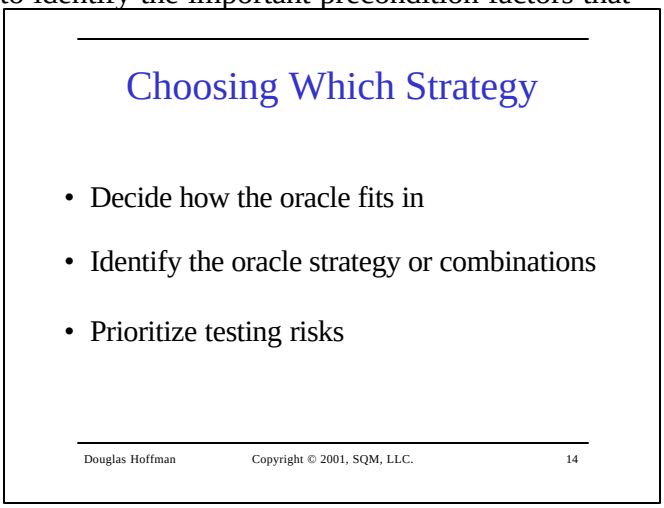
and sorting by transaction number should yield the same results. Similarly, a current employee's start date should be between their birth date and today. An employee's dependent children should be younger than the employee (but recognize that there are circumstances when the dependent may be older).



One special type of heuristic strategy uses a statistical approach based on relationships in the population statistics between inputs and results. For example, if we use uniform random numbers to generate random, symmetric, geometric figures at random locations on a page, then we might compute the mean location (X and Y coordinate) for each dot on the page. For a large number of generated figures we would expect the mean location to be the middle of the page. Likewise, we might expect the standard deviation and skew for the dots to be similar to the standard deviation and skew for the random numbers. This heuristic could allow us to create some automated tests that generate large numbers of random figures and still give us some assurance that the results are reasonable.

Choosing A Strategy

Understand what you are testing well enough to identify the important precondition factors that need to be monitored or manipulated for automated testing. Also identify the important results that need to be verified to know whether the SUT passed or failed any test. Based on the important input and result factors, decide on what oracles are needed to be able to run useful tests.



The oracle strategies will be based on availability of existing oracles, the ease of creation of new oracles, and recognition of applicable heuristics. A combination of strategies can be very effective at providing a basis for powerful automated tests.

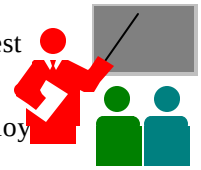
As with all testing tasks, it is important to understand the trade-offs between the various risks and the costs involved in testing. It is very easy to get lost in the wonderful capabilities of automated tests and lose sight of the important goal of releasing high quality software.

Conclusion

There are techniques to make very powerful automated tests. Test result oracles can generate predicted results or verify the correctness of actual results for automated tests. There are different types of oracles and strategies for automated verification of results. When used well, oracles can result in better, more powerful automated tests. These automated tests can be engineered from models of the SUT, its inputs and results, and test results oracles.

Summary

- Automated tests can be powerful
- Test oracles are critical factors in making good automated tests
- There are different types of test oracles available
- There are many ways to employ test oracles



Douglas Hoffman Copyright © 2001, SQM, LLC. 15

References

Hoffman, Douglas; “A Taxonomy of Test Oracles” Quality Week 1998.

Hoffman, Douglas; “Heuristic Test Oracles“ Software Testing and Quality Engineering Magazine, Volume 1, Issue 2, March/April 1999.

Hoffman, Douglas; “Test Automation Architectures: Planning for Test Automation” Quality Week 1999.

Hoffman, Douglas; “Mutating Automated Tests” STAR East, 2000.

Nyman, Noel; “Self Verifying Data - Validating Test Results Without An Oracle” STAR East, 1999.