Architectures of Test Automation

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

Professor, Department of Computer Science, Florida Institute of Technology

August 13, 2000

Acknowledgement

Many of the ideas in this presentation were jointly developed with Doug Hoffman, in a course that we taught together on test automation, and in the Los Altos Workshops on Software Testing (LAWST) and the Austin Workshop on Test Automation (AWTA).

- LAWST 5 focused on oracles. Participants were Chris Agruss, James Bach, Jack Falk, David Gelperin, Elisabeth Hendrickson, Doug Hoffman, Bob Johnson, Cem Kaner, Brian Lawrence, Noel Nyman, Jeff Payne, Johanna Rothman, Melora Svoboda, Loretta Suzuki, and Ned Young.
- LAWST 1-3 focused on several aspects of automated testing. Participants were Chris Agruss, Tom Arnold, Richard Bender, James Bach, Jim Brooks, Karla Fisher, Chip Groder, Elizabeth Hendrickson, Doug Hoffman, Keith W. Hooper, III, Bob Johnson, Cem Kaner, Brian Lawrence, Tom Lindemuth, Brian Marick, Thanga Meenakshi, Noel Nyman, Jeffery E. Payne, Bret Pettichord, Drew Pritsker, Johanna Rothman, Jane Stepak, Melora Svoboda, Jeremy White, and Rodney Wilson.
- AWTA also reviewed and discussed several strategies of test automation. Participants in the first meeting were Chris Agruss, Robyn Brilliant, Harvey Deutsch, Allen Johnson, Cem Kaner, Brian Lawrence, Barton Layne, Chang Lui, Jamie Mitchell, Noel Nyman, Barindralal Pal, Bret Pettichord, Christiano Plini, Cynthia Sadler, and Beth Schmitz.

I'm indebted to Hans Buwalda, Elizabeth Hendrickson, Alan Jorgensen, Noel Nyman , Harry Robinson, James Tierney, and James Whittaker for additional explanations of test architecture and/or stochastic testing.

Why do some groups have so much more success with test automation than others?

In 1997, Brian Lawrence and I organized a meeting of senior testers, test automators, managers, and consultants to discuss this question, forming the Los Altos Workshops on Test Automation (LAWST). Other groups (Software Test Managers Roundtable and Austin Workshop on Test Automation) have since formed along the same lines. Brian and I described the LAWST process in Kaner & Lawrence (1997, 1999).

This paper is essentially a progress report. I can't (yet) tell you how to succeed in test automation, but I know more of the pieces of more of the puzzles than I did when we started setting up the first meeting, in 1996. The progress has been uneven, and so is this paper. But I hope that it's useful.

Comments and criticisms are welcome. Please send them to me at **kaner@kaner.com**.

GUI Regression Testing is Computer Assisted Testing

GUI-level regression automation is the most common approach to test automation, but it is definitely not the only one and it has serious limitations. This paper puts this approach in context and then explores several alternatives.

As a start to the discussion, let me note that GUI-level regression automation is not automated testing. It doesn't automate very much of the testing process at all. Let's look at some of the tasks, and see who does them:

• Analyze the specification and other docs for ambiguity or other indicators of potential error	*Done by humans*
• Analyze the source code for errors	*Humans*
• Design test cases	*Humans*
• Create test data	*Humans*
• Run the tests the first time	*Humans*
• Evaluate the first result	*Humans*
• Report a bug from the first run	*Humans*
• Debug the tests	*Humans*
• Save the code	*Humans*
• Save the results	*Humans*
• Document the tests	*Humans*
• Build a traceability matrix (tracing test cases back to specs or requirements)	*Done by humans or by another tool (not the GUI tool)*
• Select the test cases to be run	*Humans*
• Run the tests	*The Tool does it*
• Record the results	*The Tool does it*
• Evaluate the results	*The Tool does it, but if there's an apparent failure, a human re-evaluates the test results.*
• Measure the results (e.g. performance measures)	*Done by humans or by another tool.*
• Report errors	*Humans*
• Update and debug the tests	*Humans*

When we see how many of the testing-related tasks are being done by people or, perhaps, by other testing tools, we realize that the GUI-level regression test tool doesn't really automate testing. It just helps a human to do the testing.

Rather than calling this "automated testing", we should call it computer-assisted testing.

I am not showing disrespect for this approach by calling it computer-assisted testing. Instead, I'm making a point—there are a lot of tasks in a testing project and we can get help from a hardware or software tool to handle any subset of them. GUI regression test tools handle some of these tasks very well. Other tools or approaches will handle a different subset. For example,

- Dick Bender's *Softtest* tool and Telcordia's *AETG* help you efficiently design complex tests. They don't code the tests, execute the tests, or evaluate results. But in some cases, the design problem is the most difficult problem in the testing project.
- Source code analyzers, like LINT or the McCabe complexity metric tool are useful for exposing defects, but they don't design, create, run or evaluate a single test case.

GUI Regression Testing has some Basic Problems

I moved to Silicon Valley in 1983, joining WordStar as Testing Technology Team Leader. Our testing was purely black box. We got code from programmers, it was broken, and we wanted to find the errors. Even though we still tested the program's compatibility with peripherals (such as printers, disk drives, keyboards, and displays) and we still paid attention to timing issues, our focus was external. Our tests looked at the program from the vantage point of an end user; our tests were all run at the user interface level.

There were tools back then to support user interface level test automation in DOS. We used ProKey and SuperKey. You could use the tool to capture your keystrokes (and later, mouse movements) as you typed. Or you could write a script yourself. The results were disappointing:

- It was reasonably quick to create a test, but it took additional time to test each test (a "script" is code—if you want to know that it works, you have to test it) and document it (you might not need any documentation if you only have 10 tests but as the numbers get higher, you need to know what you have, what you don't have, and how to interpret a failure).
- The tests were tied so closely to trivial details of the user interface that almost any change would cause the program to fail the test.
- Fixing a broken test took so long that it was as fast to just recreate the test.
- The tests didn't find many bugs, and most of the bugs that I did find this way, I found when I first created the tests.

Over the years, tools like these evolved but test groups failed too often when attempting to use them.

Looking back, I think that many of the failures were the result of weak design. It was common (and still is) to create each test case in isolation. Whether the tester captured a test case or coded it in the tool's scripting language:

- Each test case embedded constants into its code. For example, a test case involving printing might have code to pull down the *File* menu, move down 4 items to the menu item named *Print,* click on *Print* to get to the *Print Dialog*, move to the right side of the dialog to click on the *Number of Copies* field, enter *2* copies, then click on *OK* to start printing. If any print-related details of the user interface changed, the test case code had to be revised

- Tests were self-contained. They repeated the same steps, rather than relying on common, shared routines. Every test that printed had its own code to pull down the *File* menu, etc. If the user interface for printing changed, every test had to be examined and (if it prints) fixed.
- The process of coding and maintaining the tests was tedious and time-consuming. It took a lot of time to create all these tests because there was no modularity and therefore no code reuse. Rather than calling modules, every step has to be coded into every test case. Boredom results in errors, wasting even more time.

Starting in the early 1990's, several of us began experimenting with data driven designs that got around several of these problems. Bach (1996) and Kaner (1997a, 1998) discussed the problems of test automation tools in much greater detail than I cover in this paper. Buwalda (1996, 1998) described a data driven approach that is more general than the calendar example presented below. An implementation of that approach is available as *TestFrame* (check www.sdtcorp.com and Kit, 1999). Pettichord (1996, 1999) presented another general data driven approach. See also Kent (1999) and Zambelich (1998). A web search will reveal several other papers along the same lines. Pettichord's web site, www.pettichord.com, is an excellent place to start that search.

A Calendar-Based Example of Software Architecture

I owe the thinking behind this example to Brenda Takara, who laid it out for me in 1992.

Imagine testing a program that creates calendars like the ones you can buy in bookstores. Here are a few of the variables that the program would have to manage:

- The calendar might be printed in portrait (page is taller than wide) or landscape orientation.
- The calendar might have a large picture at the top or bottom or side of the page.
- If there is a picture, it might be in TIF, PCX, JPEG, or other graphics format.
- The picture might be located anywhere on the local drive or the network.
- The calendar prints the month above, below, or beside the table that shows the 30 (or so) days of the month.
- Which month?
- What language (English? French? Japanese?) are we using to name the month and weekdays?
- The month is printed in a typeface (any one available under Windows), a style (normal, italic, bold, etc.), a size (10 points? 72 points?).
- The calendar might show 7 or 5 or some other number of days per week.
- The first day of the week might be Sunday, Monday, or any other day.
- The first day of the month might be any day of the week.
- The days of the week might run from left to right or from top to bottom.
- The day names are printed in a typeface, style, or size and in a language.
- The days are shown in a table, and are numbered (1, 2, 3, etc.) Each day is in one cell (box) in the table. The number might be anywhere (top, bottom, left, right, center) in the cell.
- The cell for a day might contain a picture or text ("Mom's birthday") or both.

In a typical test case, we specify and print a calendar.

Suppose that we decided to create and automate a lot of calendar tests. How should we do it?

One way (the approach that seems most common in GUI regression testing) would be to code each test case independently (either by writing the code directly or via capture/replay, which writes the code for you). So, for each test, we would write code (scripts) to specify the paper orientation, the position, network location, and type of the picture, the month (font, language), and so on. If it takes 1000 lines to code up one test, it will take 100,000 lines for 100 tests and 1,000,000 lines for 1000 tests. If the user interface changes, the maintenance costs will be enormous.

Here's an alternative approach:

- Start by creating a table. Every column covers a different calendar attribute (paper orientation, location of the graphic, month, etc.). Every row describes a single calendar. For example, the first row might specify a calendar for October, landscape orientation, with a picture of children in costumes, 7-day week, lettering in a special ghostly typeface, and so on.
- For each column, write a routine that implements the choice in that column. Continuing the October example, the first column specifies the page orientation. The associated routine provides the steps necessary to set the orientation in the software under test. Another routine reads "October" in the table and sets the calendar's month to October. Another sets the path to the picture. Perhaps the average variable requires 30 lines of code. If the program has 100 variables, there are 100 specialized routines and about 3000 lines of code.
- Finally, write a control program that reads the calendar table one row at a time. For each row, it reads the cells one at a time and calls the appropriate routine for each cell.

In this setup, every row is a test case. There are perhaps 4000 lines of code counting the control program and the specialized routines. To add a test case, add a row. Once the structure is in place, additional test cases require 0 (zero) additional lines of code.

Note that the table that describes calendars (one calendar per row) is independent of the software under test. We could use these descriptions to test any calendar-making program. Changes in the software user interface won't force maintenance of these tests.

The column-specific routines will change whenever the user interface changes. But the changes are limited. If the command sequence required for printing changes, for example, we change the routine that takes care of printing. We change it once and it applies to every test case.

Finally, the control program (the interpreter) ties table entries to column-specific routines. It is independent of the calendar designs and the design of the software under test. But if the scripting language changes or if we use a new spreadsheet to store the test cases, this is the main routine that will change.

In sum, this example reduces the amount of repetitious code and isolates different aspects of test descriptions in a way that minimizes the impact of changes in the software under test, the subject matter (calendars) of the tests, and the scripting language used to define the test cases. The design is optimized for maintainability.

Maintainability is a Requirements Issue

Many attempts to use GUI-level regression tools failed. The maintenance problem was one of the core reasons. The programmers delivered code, the testers developed automated and manual tests, they found bugs and complained about the program's user interface. The programmers made changes, revising the design and adding features. But now the test cases had to be revised because the UI had changed. After several cycles of this, the testers found that they were spending too much of their time fixing the test code and too little of their time finding and reporting bugs.

It is easy (and it was popular in some circles) to dismiss these failures by saying that the programmers should stabilize the design early. The programmers should do some form of "real" engineering, using something like the waterfall model or the V model. They should lock down the external design before writing any (or much) code. If those darned programmers would just follow these good practices, then the code they gave to testers wouldn't go through many late design changes and maintenance would be much less of a problem.

Maybe, if we ignored the importance of reusing the code in the next release (when the design will assuredly change), this dismissal would make some sense. If only those programmers would go along and change their approach. But what if they won't change their approach to development? What if they think that incremental design and development is part of the strength of their process, not a weakness. In that case, saying that the programmers should work differently is an excuse for failure.

Rather than treating late design changes as an excuse for failure, we can recharacterize the problem. If the programmers *will* make late changes, *then it must be a requirement of the automation to deal smoothly with late changes*. Under that requirement, we would want to design a series of tests that are easy to maintain.

A data driven design, if done well, is an example of design to reduce the maintenance effort.

Test automation is software development. Like all software under development, it makes a lot of sense to figure out your requirements and create something that meets them.

Gause & Weinberg (1989) present a useful series of context-free questions (questions that are useful under a wide range of circumstances) for discovering a project's requirements. Michalko (1991) presents another set that were developed by the CIA to analyze a problem and evaluate proposed solutions. Lawrence & Gause (1998) provide additional questions and checklists.

At the LAWSTs, we listed 27 questions that some of us use to discover requirements for a test automation project. (None of us uses all of the questions—this is a collection of several different people's work.) I hope that the list that follows is largely self-explanatory. For additional discussion, see Kaner (1998).

1. Will the user interface of the application be stable or not?
2. To what extent are oracles available?
3. To what extent are you looking for delayed-fuse bugs (memory leaks, wild pointers, etc.)?
4. Does your management expect to recover its investment in automation within a certain period of time? How long is that period and how easily can you influence these expectations?
5. Are you testing your own company's code or the code of a client? Does the client want (is the client willing to pay for) reusable test cases or will it be satisfied with bug reports and status reports?
6. Do you expect this product to sell through multiple versions?
7. Do you anticipate that the product will be stable when released, or do you expect to have to test Release N.01, N.02, N.03 and other bug fix releases on an urgent basis after shipment?
8. Do you anticipate that the product will be translated to other languages? Will it be recompiled or relinked after translation (do you need to do a full test of the program after translation)? How many translations and localizations?
9. Does your company make several products that can be tested in similar ways? Is there an opportunity for amortizing the cost of tool development across several projects?
10. How varied are the configurations (combinations of operating system version, hardware, and drivers) in your market? (To what extent do you need to test compability with them?)

11. What level of source control has been applied to the code under test? To what extent can old, defective code accidentally come back into a build?
12. How frequently do you receive new builds of the software?
13. Are new builds well tested (integration tests) by the developers before they get to the tester?
14. To what extent have the programming staff used custom controls?
15. How likely is it that the next version of your testing tool will have changes in its command syntax and command set?
16. What are the logging/reporting capabilities of your tool? Do you have to build these in?
17. To what extent does the tool make it easy for you to recover from errors (in the product under test), prepare the product for further testing, and re-synchronizethe product and the test (get them operating at the same state in the same program).
18. (In general, what kind of functionality will you have to add to the tool to make it usable?)
19. Is the quality of your product driven primarily by regulatory or liability considerations or by market forces (competition)?
20. Is your company subject to a legal requirement that test cases be demonstrable?
21. Will you have to be able to trace test cases back to customer requirements and to show that each requirement has associated test cases?
22. Is your company subject to audits or inspections by organizations that prefer to see extensive regression testing?
23. If you are doing custom programming, is there a contract that specifies the acceptance tests? Can you automate these and use them as regression tests?
24. What are the skills of your current staff?
25. Do you have to make it possible for non-programmers to create automated test cases?
26. To what extent are cooperative programmers available within the programming team to provide automation support such as event logs, more unique or informative error messages, and hooks for making function calls below the UI level?
27. What kinds of tests are really *hard* in your application? How would automation make these tests easier to conduct?

GUI Regression is Limited Even if Done Well

Regression testing has inherent limits. Even if you do it well, the key problem is that you keep running the same tests, time after time. Maybe the test found a bug the first time or two that you ran it, but eventually, in most programs, the tests stop finding bugs. That's not because there are no bugs left in the program. It's because there are no bugs left that you can find by rerunning the same old tests. To find new bugs, it helps to run tests that you haven't tried before.

It makes some people uncomfortable when I criticize regression testing because they have had good experiences. Before I continue, let me acknowledge some of the circumstances under which this repetitive testing makes good sense:

- ***Smoke testing:*** This is a standard, relatively brief test suite. Every time there's a new build, these tests are run to check whether there is anything fundamentally wrong with the software. If so, the build is rejected. Only builds that pass the smoke test get more thorough testing.
- ***Configuration testing:*** If you have to run the same test across 50 printers, you recover your investment in the computer-assisted tests the first night you use it to run the tests.

- ***Variations on a theme:*** The regression framework, generalized slightly, can make it easy to cheaply run a few hundred similar tests. If you need to vary a parameter in small increments across a range, it might be convenient to code a single regression test with a variable (the parameter under test) that you can feed different values to.
- ***Weak configuration management or fragile code:*** Sometimes, changes do break the same code in essentially the same ways, over and over again. If so, and if you can't control the root cause of this problem, a standard series of relevant tests will be fruitful.
- ***Multiple causes yield the same symptoms:*** Some programs are designed with minimal user interfaces. Many functions feed to the same display routines. If anything goes wrong in any of them, you get what looks like the same failure. The same tests might yield what looks like the same failure for many different reasons.
- ***Performance benchmarking:*** Do exactly the same thing across many builds or many systems in order to discover changes or differences in performance.
- ***Demonstration to regulators:*** In regulated industries, it might be useful (or required) to have a series of standard tests that check each of the claims made for the software and that check for handling of common bad inputs or device errors. The software company might do a huge amount of additional testing, but people developing software that is regulated by the U.S. Food and Drug Administration have told me that the core suite of tests is essential for the FDA.

Having paid my respects to the method, my question is whether there are alternatives that can cost-effectively help us discover new defects. There are several such alternatives.

A Classification Scheme for Automated Tests

We can describe many types of automated tests in terms of the following dimensions:

- Source of test cases
  - Old
  - Ongoing monitoring
  - Intentionally designed, new
  - Random new
- Size of test pool
  - Small
  - Large
  - Exhaustive
- Evaluation strategy
  - Comparison to saved result
  - Comparison to an oracle
  - Comparison to a computational or logical model
  - Comparison to a heuristic prediction. (NOTE: All oracles are heuristic.)
  - Crash
  - Diagnostic

    - State model
- Serial dependence among tests
    - Independent
    - Sequence is relevant

There's at least one other key dimension, which I tentatively think of as "level" of testing. For example, the test might be done at the GUI or at the API level. But this distinction only captures part of what I think is important. Testing at the GUI is fine for learning something about the way the program works with the user, but not necessarily for learning how the program works with the file system or network, with another device, with other software or with other technical aspects of the system under test. Nguyen (2000) and Whittaker (1998) discuss this in detail, but I haven't digested their discussions in a way that I can smoothly integrate into this framework. (As I said at the start of the paper, this is a progress report, not a finished product.)

Within this scheme, GUI level regression tests look like this:

- Source of tests: *Old (we are reusing existing tests).*
- Size of test pool: *Small (there are dozens or hundreds or a few thousand tests, not millions).*
- Evaluation strategy: *Comparison to a saved result (we expect the software's output today to match the output we got yesterday).*
- Serial dependence among tests: *Independent (our expectations for this test don't depend on the order in which we ran previous tests).*

Let's consider some other examples.

Exhaustive High-Volume Testing

Doug Hoffman (personal communication) did testing of mathematical functions on the MasPar (massively parallel) computer. The computer had 65,000 parallel processors. One of the functions he tested was integer square root. The word size on this computer was 32 bits, so there were 2^32 (4,294,967,296) different integers. The computer was expected to be used for life-critical applications, so the company was willing to do extensive testing.

The test team wrote an additional square root function, which they used as an oracle. (I define an *oracle* as a program that you trust more than the software under test, that you can use to determine whether results from the software under test are probably incorrect.)

To test the square roots of all of the 32-bit integer required a total of about 6 minutes. In the course of doing this exhaustive (check all values) test, Hoffman found two errors that were not associated with any boundary or with any obvious special value for checking calculations. (It turned out that a bit was sometimes mis-set, but the incorrect value of that bit was irrelevant in all but two calculations.)

If the team had not done exhaustive testing, they would probably have missed these bugs.

Classification:

- Source of tests: *Intentionally designed, new (we are generating new tests every time, but each test is intentionally designed, rather than being created using a random number generator).*
- Size of test pool: *Exhaustive (every value is tested).*
- Evaluation strategy: *Comparison to an oracle.*
- Serial dependence among tests: *Independent.*

Basic Random Testing: Function Equivalence Testing

Hoffman's next task involved 64-bit integer square roots. To test all of these would have required 2^32 times as many tests as the 32-bit tests, or about 4 million hours. Exhaustive testing was out of the question, so the group did a huge random sample instead.

Function equivalence testing refers to a style of testing in which a function under test is compared (under a wide range of values) with an oracle, a function that it is supposed to be equivalent to.

Classification:

- Source of tests: *Random, new (the value that is tested is determined by a random number generator).*
- Size of test pool: *Large (billions of tests).*
- Evaluation strategy: *Comparison to an oracle.*
- Serial dependence among tests: *Independent.*

Several functions can be used as oracles. For example, we might:

- Compare the new Release 3.0's behavior with functions that we knew worked in the old Release 2.0.
- Compare our product with a competitor's.
- Compare our product with a standard function (perhaps one loaded off the net, or coded based on explicit instructions in a textbook).
- Use our function with its inverse (a mathematic inverse like squaring a square root or an operational one, such as splitting a merged table).
- Take advantage of a known relationships (such as $\text{sine}^2(x) + \text{cosine}^2(x) = 1$.)
- Use our product to feed data or messages to a companion product (that we trust) that expects our output to be formatted in certain ways or to have certain relationships among fields.

For a thorough discussion of random numbers, see Donald Knuth's *Art of Computer Programming*, Volume 2. Or for a simpler presentation, Kaner & Vokey (1982).

The beauty of random testing (whether we use sequentially related or independent tests) is that we can keep testing previously tested areas of the program (possibly finding new bugs if programmers added them while fixing or developing some other part of the system) but we are always using new tests (making it easier to find bugs that we've never looked for before). By thoughtfully or randomly combining random tests, we can create tests that are increasingly difficult for the program, which is useful as the program gets more stable. The more extensive our oracle(s), the more we can displace planned regression testing with a mix of manual exploratory testing and computer-assisted random tests.

Stochastic Tests Using Dumb Monkeys

A stochastic process involves a series of random events over time. The distinction between a stochastic process and generic random testing is that random tests may or may not take into account sequence. For example, function equivalence testing (testing one function for equivalence with an oracle) can be done with random data but it is not stochastic. We care about the comparisons one by one but don't care about the order of comparisons.

Markov processes are a simple example of a stochastic process. The Markov process is defined as follows: If En, En+1, En+2, etc. are events n, n+1, n+2 (etc) in a sequence, and if P(En+2 | En+1) is the conditional probability that En+2 will be the next event, given that the current event is En+1, then for all

possible events, P(En+2 | En+1 and En) = P(En+2 | En+1). That is, the sequential dependence is between the current state and the next one, only. How we got to where we are is irrelevant. In the more general stochastic process, the conditional dependence might go back any number of states.

A stochastic test involves a random series of test cases. The individual tests might be randomly designed or intentionally designed, but the order is random and there is a belief (of the test designer) that the order is relevant. Stochastic tests might help us find memory leaks, stack corruption, memory corruption caused by wild pointers, or other misbehaviors that involve a delayed or gradual reaction to a software failure.

A dumb monkey is a stochastic test that has no model of the software under test (Nyman, 1998). The test tool generates random inputs to the software under test. Some of them correspond to valid commands, others yield error messages.

The monkey can be refined by restricting its behavior (block it from using the Format command or from sending e-mail messages) or by giving it some knowledge of the environment in which the software is running. For example, if you press a button in the application, the graphic associated with that button should change in a predictable way. The monkey might not know what the button does, but it might be able to recognize that the button's graphic hasn't changed or has changed inappropriately.

Noel Nyman used techniques like this as part of the testing of Windows NT's compatibility with a large number of applications.

Dumb monkeys are useful early in testing. They are cheap to create and, while the software is unstable, they can reveal a lot of defects. Note, though, that the software can't be considered stable just because it can run without crashing under a dumb monkey for several days. The program might be failing in other critically important ways (such as corrupting its data), without crashing.

Classification:

- Source of tests: *Random, new.*
- Size of test pool: *Large.*
- Evaluation strategy: *Crash or light diagnostics.*
- Serial dependence among tests: *Sequence is relevant.*

Stochastic Tests Using Diagnostics

In Kaner (1997b), I described a malfunctioning telephone system in some detail. In short, the system had a subtle stack corruption bug. Phones would go out of service if, without passing through the most common state in the system, they passed through an uncommon state more than 10 times, and attempted to put a 10th item on a stack.

In this case, all of the individual functions appeared to work, simple testing of all lines and branches in the program would not (and didn't) reveal the problem, and until the phone crashed, there were no symptoms that a black box tester could notice.

Eventually, the programming team and the testing team(s) (hardware and software testers) collaborated on a simulator-based strategy. The simulator generated random events. Whatever state the phone was in, the simulator might give it any input. The testers could adjust the simulation in order to increase or decrease the frequency of use of certain features. The programmers added diagnostics to the code. If the testers were focusing on the hold feature this week, the programmers would add a large number of hold-related diagnostics. When the testers shifted to another feature (like forwarding), the programmers would compile out the hold diagnostics and add in diagnostics for forwarding.

The simulator generated long chains of randomly chosen events. Some of these would trigger a diagnostic (reporting an unexpected state transition, or a bad value on a stack, etc.) The diagnostic output would be

logged to a printer, with supporting detail to help the tester or the programmer recognize and replicate the failure.

In the telephone case, the programmers conditionally compiled additional diagnostics into the code. In other cases, the code or the hardware has permanently-built-in diagnostics. For example, a testing program might take advantage of system-available commands in a device under test, to check for free memory or for the state of attached devices. In the firmware embedded into some devices, there might be dozens or several hundred diagnostic commands available to the programmer and that could be available to the testers. So, for example, if we had access to built-in diagnostics, we might use this style of testing for the software that controls cars' fuel injectors, or simple medical devices, or cash registers.

One challenge that is outside of the scope of this report is the reproducibility problem. For example, if the program fails after a random series of 20,000 tests, how do we know what sub-sequence is necessary (or sufficient) to replicate the defect?

Classification:

- Source of tests: *Random, new.*
- Size of test pool: *Large.*
- Evaluation strategy: *Diagnostics (or crash).*
- Serial dependence among tests: *Sequence is relevant.*

Stochastic Tests Using a State Model

For any state, you can list the actions that the user can take, and the expected result of each action (what new state the program should reach, and what check the test could perform in order to determine whether the program actually made the transition to the new state.) In model-based testing, the tester creates a state model of the system and then runs a large set of tests to walk the state model.

Typically, model-based testing is Markovian (we think in terms of transitions from the current state to the next one, not in terms of sequences of three or four or five states), but the series tested might involve hundreds or thousands of state transitions (mini-test cases).

For further information on this approach, see Jorgensen (1999), Jorgensen & Whittaker (2000), Robinson (1999a, 1999b), and Whittaker (1997).

Classification:

- Source of tests: *Random, new.*
- Size of test pool: *Large.*
- Evaluation strategy: *State model (conformance with the model).*
- Serial dependence among tests: *Sequence is relevant.*

A Heuristic Test

Suppose that you were running an online retail system. Suppose further that over the last year, almost all of your sales were in your home state (Florida) and almost none of your customers were from Wyoming. But today, you had a big increase in volume and 90% of your sales were from Wyoming. This might or might not indicate that your code has gone wild. Maybe you just got lucky and the State of Wyoming declared you a favored vendor. Maybe all that new business is from state agencies that never used to do business with you. On the other hand (and more likely), maybe your software has gone blooey.

In this case, we aren't actually running tests. Instead, we are monitoring the ongoing behavior of the software, comparing the current results to a statistical model based on prior performance.

Additionally, in this case, our evaluation is imperfect. We don't know whether the software has passed or failed the test, we merely know that its behavior is highly suspicious. In general, a heuristic is a rule of thumb that supports but does not mandate a conclusion. We have partial information that will support a probabilistic evaluation. This won't tell you for sure that the program works correctly, but it can tell you that the program is *probably* broken.

One of the insights that I gained from Doug Hoffman is that all oracles can be thought of as heuristic oracles. Suppose that we run a test using an oracle—for example, comparing our program's square root function with an oracle. The two programs might agree that the square root of 4 is 2, but there's a lot of other information that isn't being checked. For example, what if our program takes an hour to compute the square root? Or what if the function has a memory leak? What if it sends a message to cancel a print job? When we run a test of our software against an oracle, we pay attention to certain inputs and certain outputs, but we don't pay attention to others. For example, we might not pay attention to the temperature (recently, I saw a series of failures in a new device that were caused by overheating of the hardware. They *looked like* software errors, and they were an irreproducible mystery until someone noticed that testing of the device was being done in an area that lacked air conditioning and that the device seemed very hot.) We might not pay attention to persistent states of the software (such as settings of the program's options) or to other environmental or system data. Therefore, when the program's behavior appears to match an oracle's, we don't know that the software has passed the test. And if the program fails to match the oracle, that might be the result of an error of it might be a correct response to inputs or states that we are not monitoring. We don't fully know about pass or fail. We only know that there is a likelihood that the software has behaved correctly or incorrectly.

Using relatively simple heuristic oracles can help us inexpensively spot errors early in testing or run a huge number of test cases through a simple plausibility check.

Classification of this example (the retail system):

- Source of tests: *Ongoing monitoring.*
- Size of test pool: *Large.*
- Evaluation strategy: *Comparison to a heuristic prediction.*
- Serial dependence among tests: *Sequence might be relevant.*

A Scheme for Evaluating Types of Automated Tests

Doug Hoffman and I have been appraising different types of test automation strategies by asking the following question:

> What characteristics of the
>
> - Goal of testing
> - Level of testing (e.g. UI, API, unit, system)
> - Software under test
> - Environment
> - Generator of test cases
> - Reference function
> - Evaluation function
> - Users

- Risks

would support, counter-indicate, or drive you toward using the strategy under consideration?

I can't provide an analysis for each of the automation strategies, but here is an example of the approach.

Suppose that we were evaluating GUI-level regression testing. Here are some favorable conditions. Please note that factors that are favorable to one strategy might also favor another, and that they might or might not be necessary or sufficient to convince us to use that strategy. Additionally, these are just examples. You might think of other factors that would be favorable to the strategy that I haven't listed.

- ***Goal of testing:*** It would make sense to use a GUI-level regression tool if our goal was
  - Smoke testing
  - Port testing (from code written for one O/S, ported to another, for example)
  - Demonstrate basic functionality to regulators or customers
  - Ability to quickly check patch releases
- ***Level of testing (e.g. UI, API, unit, system):*** A GUI tool will test at the GUI.
- ***Software under test:***
  - For a GUI-based regression test under Windows, we're more likely to succeed if the developers used standard controls rather than custom controls. The test code dealing with custom controls is more expensive and more fragile.
  - If we were willing to drop down a level, doing regression testing at the API level (i.e. by writing programs that called the API functions), the existence of a stable set of API's might save us from code turmoil. In several companies, the API stays stable even when the visible user interface chances daily.
  - The more stable the design, the lower the maintenance cost.
  - The software must generate repeatable output. As an example of a problem, suppose the software outputs color or gray-scale graphics to a laser printer. The printer firmware will achieve the color (or black/white) mix by dithering, randomly placing dots, but in an appropriate density. Run the test twice and you'll get two outputs that might look exactly the same but that are not bit-for-bit the same.
- ***Environment:***
  - Some embedded systems give non-repeatable results.
  - Real-time, live systems are usually not perfectly repeatable.
- ***Generator of test cases***. The generator is the tool or method you use to create tests. You want to use a test strategy like regression, which relies on a relatively small number of tests, if:
  - It is expensive to run tests in order to create reference data, and therefore it is valuable to generate test results once and use them from the archives.
  - It is expensive to create the tests for some other reason.
- ***Reference function:*** This is the oracle or the collection of saved outputs. Compare the results to the output from your program under test. One could argue that it favors the strategy of regression automation if:

    - Screens, states, binary output, or a saved database are relatively easy to get and save.
    - Incidental results, such as duration of the operation, amount of memory used, or the exiting state of registers are easy to capture and concise to save.
- ***Evaluation function:*** This is the method you use to decide whether the software under test is passing or failing. For example, if you compare the current output to a previous output, the evaluation function is the function that does the actual comparison.
    - I don't see anything that specifically favors GUI regression testing.
- ***Users:*** Who will use the software? Who will use the automation?
    - Non-programmers are common users of these tools and frequent targets of vendors of these tools. To the extent that you have a sophisticated user, the simple old/new comparison might not be very informative.
- ***Risks:*** What risks are we trying to manage by testing?
    - We shouldn't be thinking in terms of (large sample) exhaustive or high-volume regression testing.

You might find it useful to build a chart with this type of information, for each testing strategy that you are considering. My conjecture is that, if you do this, you will conclude that different potential issues in the software are best tested for in different ways. As a result, you might use a few different automation strategies, rather than settling into reliance on one approach.

References

Bach, J. (1996) "Test Automation Snake Oil", *Windows Tech Journal*, October. Available at http://www.satisfice.com/articles/test_automation_snake_oil.pdf.

Buwalda, H. (1996) "Automated testing with Action Words, Abandoning Record & Playback"; Buwalda, H. (1998) "Automated testing with Action Words", *STAR Conference West*. For an accessible version of both of these, See Hans' chapter in Fewster, M. & Graham, D. (2000) Software Test Automation : Effective Use of Test Execution Tools.

Gause, D. & Weinberg, G. (1989) *Exploring Requirements : Quality Before Design*.

Jorgensen, A. (1999), *Software Design Based on Operational Modes*, Doctoral dissertation, Florida Institute of Technology.

Jorgensen, A. & Whittaker, J. (2000), "An API Testing Method", *Star Conference East*. Available at www.aet-usa.com/STAREAST2000.html.

Kaner, C. & Vokey, J. (1982), "A Better Random Number Generator for Apple's Floating Point BASIC", *Micro*. Available by e-mail from kaner@kaner.com.

Kaner, C. (1997a), "Improving the Maintainability of Automated Tests", *Quality Week*. Available at www.kaner.com/lawst1.htm.

Kaner, C. (1997b), "The Impossibility of Complete Testing", *Software QA*. Available at www.kaner.com/imposs.htm.

Kaner, C. & Lawrence, B. (1997) "Los Altos Workshop on Software Testing", www.kaner.com/lawst.htm.

Kaner, C. (1998), "Avoiding Shelfware: A Manager's View of Automated GUI Testing", STAR East, 1998; available at www.kaner.com or from the author (kaner@kaner.com).

Kaner, C. & Lawrence, B. (1999), *The LAWST Handbook*, available from the authors (kaner@kaner.com; brian@coyotevalley.com).

Kent, J. (1999) "Advanced Automated Testing Architectures", Quality Week.

Kit, E. (1999) "Integrated, Effective Test Design and Automation", *Software Development*, February issue. Available at www.sdmagazine.com/breakrm/features/s992f2.shtml.

Lawrence, B. & Gause, D. (1998), *Gause & Lawrence Requirements Tools*. Available at www.coyotevalley.com/stuff/rt.doc.

Michalko, M. (1991), *Thinkertoys (A Handbook of Business Creativity)*. (See especially the Phoenix Questions at p. 140).

Nguyen, H. (2000), *Testing Applications on the Web: Test Planning for Internet-Based Systems*, manuscript in press with John Wiley & Sons.

Nyman, N. (1998), "Application Testing with Dumb Monkeys", *STAR Conference West.*

Pettichord, B. (1996) "Success with Test Automation", *Quality Week*. Available at www.io.com/~wazmo/succpap.htm (or go to www.pettichord.com).

Pettichord, B. (1999) "Seven Steps to Test Automation Success", STAR West, 1999. Available (later version) at www.io.com/~wazmo/papers/seven_steps.html (or go to www.pettichord.com).

Robinson, H. (1999a), "Finite State Model-Based Testing on a Shoestring", *STAR Conference West*. Available at www.geocities.com/model_based_testing/shoestring.htm.

Robinson, H. (1999b), "Graph Theory Techniques in Model-Based Testing", *International Conference on Testing Computer Software*. Available at www.geocities.com/model_based_testing/model-based.htm.

Whittaker, J. (1997), "Stochastic Software Testing", *Annals of Software Engineering*, 4, 115-131.

Whittaker, J. (1998), "Software Testing: What it is and Why it is So Difficult", available at www.se.fit.edu/papers/SwTestng.pdf.

Zambelich, K. (1998), "Totally Data-Driven Automated Testing", www.sqa-test.com/w_paper1.html.

