Avoiding Shelfware:
A Managers' View of Automated GUI Testing

Summary

Companies abandon automation efforts after failing to achieve bug-effective and cost-effective automation. Success is possible but far from assured. This paper considers issues that managers and executives should consider in the design and staffing of test automation projects. The paper's considers:

- The GUI-Level Automated Testing Paradigm
- Nineteen Common Mistakes
- Twenty-Seven Questions About Requirements
- Planning for Short-Term and Long-Term Returns on Investment
- Six Successful Architectures
- Conclusions Reached in the Los Altos Workshop on Software Testing
- Structure of the Los Altos Workshop on Software Testing

Automated black box, GUI-level regression test tools are popular in the industry. According to the popular mythology, people with little programming experience can use these tools to quickly create extensive test suites. The tools are (allegedly) easy to use. Maintenance of the test suites is (allegedly) not a problem. Therefore, the story goes, a development manager can save lots of money and aggravation, and can ship software sooner, by using one of these tools to replace some (or most) of those pesky testers.

Unfortunately, many attempts to use GUI-level test automation tools fail. This shouldn't be surprising. A test automation project is a software development project. As with any software development effort, if we use poor requirements, design, and coding practices, we will get bad results.

This paper raises concerns and suggestions that I raise when I help companies manage an automated testing effort. I don't pretend to be an expert on current tools—it's been ten years since I wrote any of my own code. But I do have experience in automation programming and in managing or consulting to folk who are automating.

Several of the insights in this paper are gleaned from the Los Altos Workshops on Software Testing (LAWST), a series of professionally facilitated 2-day discussions by 15-20 senior testers and testing consultants, focused on automated testing. Appendix A will describe these workshops in more detail.The material at LAWST is developed jointly. We agreed that when any of us publishes material from LAWST, he or she will list the other attendees.

The participants in the first LAWST (which focused on automation maintainability) were Chris Agruss (Autodesk), Tom Arnold (ST Labs), James Bach (ST Labs), Jim Brooks (Adobe Systems, Inc.), Doug Hoffman (Software Quality Methods), Cem Kaner (kaner.com), Brian Lawrence (Coyote Valley Software Consulting), Tom Lindemuth (Adobe Systems, Inc.), Brian Marick (Testing Foundations), Noel Nyman (Microsoft), Bret Pettichord (Unison), Drew Pritsker (Pritsker Consulting), and Melora Svoboda (Electric

Communities). Organizational affiliations are given for identification purposes only. Participants' views are their own, and do not necessarily reflect the views of the companies listed.

The participants in the second LAWST (which focused on requirements for automation) were Chris Agruss, Tom Arnold, James Bach, Richard Bender (Richard Bender & Associates), Elizabeth Hendrickson (Quality Tree Consulting), Doug Hoffman, Keith W. Hooper, III (Systemodels), Cem Kaner, Brian Lawrence, Thomas Lindemuth, Brian Marick, Noel Nyman, Bret Pettichord, Drew Pritsker, and Melora Svoboda.

The participants in the third LAWST (which focused on test documentation) were Chris Agruss, James Bach (SmartPatents), Karla Fisher (Intel), David Gelperin (Software Quality Engineering), Chip Groder (Cadence Design Systems), Elisabeth Hendrickson, Doug Hoffman, III, Bob Johnson, Cem Kaner, Brian Lawrence, Brian Marick, Thanga Meenakshi (Net Objects), Noel Nyman, Jeffery E. Payne (Reliable Software Technologies), Bret Pettichord (Tivoli Software), Johanna Rothman (Rothman Consulting Group), Jane Stepak, Melora Svoboda (Facetime), Jeremy White (CTB/McGraw-Hill), and Rodney Wilson (Migration Software Systems).

The GUI-Level Automated Testing Paradigm

Here is the basic paradigm for GUI-based automated regression testing:

(a) Design a test case, then run it.

(b) If the program fails the test, write a bug report. Start over after the bug is fixed.

(c) If the program passes the test, automate it. Run the test again (either from a script or with the aid of a capture utility). Capture the screen output at the end of the test. Save the test case and the output.

Next time, run the test case and compare its output to the saved output. If the outputs match, the program passes the test.

A reader of an earlier version of this paper said to me that this is a flawed paradigm ("a straw paradigm"). I agree with him that it is flawed—badly flawed. But if this is a recipe for failure, we should understand that it is a recipe that many of us were taught to cook with when testing GUI-based applications in Windows or Mac environments. And time after time, when I teach courses on testing, students who work in those worlds tell me that they (instead of the product they were testing) were cooked with it. So, this is the one that I'll confront in this paper.

There are many other ways to automate. I'll suggest a few in this paper, but there are lots of others. Many of them don't involve the software's user interface at all. Many of my comments don't apply to those other situations. Be thoughtful about generalizing the points made here.

Nineteen Common Mistakes

A few years ago, a colleague and I were hoping to convince our employer's senior management to buy automated testing software for us. One vendor's representatives came to our site and showed us just how cheap, quick and easy it would be to use this tool to create and run test cases.

He created test cases using capture/replay. He said that we could capture test cases as we tested this way, so we'd spend almost no extra time on test case design or test case creation.

We protested politely that these test cases wouldn't hold up if our user interface changed. He agreed. He said that programmers had to be disciplined, that they had to write external specifications and stick to them. It would be good for us to clean up our development processes.

This was a smooth-talking, persuasive salesman. He did a good job of making his tool sound cost effective, and he planted some clear expectations in people's minds.

Let's look at the points he made, in turn:

- He demonstrated capture/replay. It looks quick and cheap, but (as we'll see below) it is not a wise way to create test cases.
- He just started creating test cases. No planning phase, no requirements analysis, no design phase, none of that overhead stuff. Since when does automation of anything not require a disciplined approach to software development? Of all people, testers should realize that quick-and-dirty design and implementation will lead us to fail as miserably as so many of the applications we have tested.
- It was really quick to create these test cases. It isn't anywhere near this quick in the real world. Not if you want to keep the test and reuse it.
- He created simple cases that didn't require any thinking. He didn't create any documentation for them. He didn't test them.
- He made the tests standalone—they shared no common code. A simple change in the UI could force you to recode or recapture every single test case, rather than requiring only one comparably simple change in the automation code.
- His preaching about the waterfall model didn't fit our environment and never would. This appeal to The One True Development Model is pretty effective. It instills guilt or shame in people that they are not doing development The Right Way. Many executives are quick to make pious pronouncements that hereinafter we will really, truly, always follow the waterfall. But that doesn't mean that the company will actually work this way. Ours certainly wasn't going to (nor has any company that I've worked for or with, in my 15 years in software development in Silicon Valley).

This salesman made a persuasive case that his tool would be cheap, easy, and effective. But even if it was a great product, we could never have lived up to the promises this person was making for us to our management. We would have been failures, no matter how good a job we did. Fortunately, we had the good sense to talk our management out of buying this product.

This salesman's presentation encouraged us to make several classic mistakes. There are many pitfalls in automated regression testing. Here's a longer list of them. See also Bach (1995).

1. Don't underestimate the cost of automation.

It takes time to automate a test. It takes about 3-10 times as long (and can take much longer) to create, verify, minimally document,[1] and save an individual automated test this way as it does to create and execute a manual test once. If you are testing a Windows program that uses custom controls, this ratio might be more than 30 to 1. Many tests will be worth automating, but for all the many tests that you run only once or twice, this approach is inefficient.

There are ways to reduce the unit cost of automated tests, but they require substantial up-front planning.

[1] A reader suggested that this is an unfair comparison. *If we don't count the time spent documenting manual tests, why count the time spent documenting the automated tests?* In practice, there is a distinction. A manual test can be created once, to be used right now. You will never reuse several of these tests; documentation of them is irrelevant. An automated test is created to be reused. You take significant risks if you re-use a battery of tests without having any information about what they cover.

You don't gain back much of that time during the Release (such as a product Release 2.0) in which you do the initial automation programming. If you are going to achieve your usual level of testing, you have to add staff. If a project would normally take ten testers one year for manual testing, and you are going to add two programmer-years of automation work, then to find the same number of bugs, you will have to keep the ten testers and add two programmers. In the next Release, you might be able to cut down on tester time. In this Release, you'll save some time on some tasks (such as configuration testing) but you'll lost some time on additional training and administrative overhead. By the very end of the project, you might have improved your ability to quickly regression test the program in the face of late fixes, but at this last-minute point in the schedule, this probably helps you test less inadequately, rather than giving you an opportunity to cut staffing expense.

NINETEEN COMMON MISTAKES

1. Don't underestimate the cost of automation.
2. Don't underestimate the need for staff training.
3. Don't expect to be more productive over the short term.
4. Don't spend so much time and effort on regression testing.
5. Don't use instability of the code as an excuse.
6. Don't put off finding bugs in order to write test cases.
7. Don't write simplistic test cases.
8. Don't shoot for "100% automation."
9. Don't use capture/replay to create tests.
10. Don't write isolated scripts in your spare time.
11. Don't create test scripts that won't be easy to maintain over the long term.
12. Don't make the code machine-specific.
13. Don't fail to treat this as a genuine programming project.
14. Don't "forget" to document your work.
15. Don't deal unthinkingly with ancestral code.
16. Don't give the high-skill work to outsiders.
17. Don't insist that all of your testers be programmers.
18. Don't put up with bugs and crappy support for the test tool.
19. Don't forget to clear up the fantasies that have been spoonfed to your management.

2. Don't expect to increase productivity over the short term.

When GUI-level regression automation is developed in Release N of the software, most of the benefits are realized during the testing and development of Release N+1. We were surprised at the first LAWST to realize that we all shared this conclusion, because we are so used to hearing about (if not experiencing) the oh-so-fast time to payback for an investment in test automation. Some things do pay back the investment quickly (see below), but if you automate extensively, you'll automate many tests that you wouldn't otherwise have run ten times during Release N. You are thus spending more time on each of these tests than you would have spent on them if you were testing manually during this Release cycle.

3. Don't underestimate the need for staff training.

These tools come with their own funky programming languages, their own custom debugging tools and other support tools, their own jargon, and their own vision about the best way to do product development.

They also come with limited documentation. There are few third party books like Tom Arnold's (1996).

If your staff were all Pascal programmers and your VP ordered you to henceforth do all programming in C++, how long would it take you to become expert programmers in the new language and programming environment? Product Development executives understand the learning curve issue here and so they should understand your problem when you explain that you face the same issue with the new tool and its new language. You need time to learn the tool. Plan for it.

4. Don't spend so much time and effort on regression testing.

Why do we want to do regression testing? How much value do we get out of regression testing? Once a program has passed a test, how likely is this test to find a bug in the future?

In last year's STAR Conference, Chip Groder (1997) provided data on the number of bugs he found during development of automated test cases compared to the number he found once the test cases were being executed. Over 88% of the bugs were found during the manual testing that was done as part of the development of the tests. Comparable results were reported at LAWST.

If a test case is unlikely to find bugs, what is the point of running it? Some people say, "It gives us confidence that the program is still OK." But isn't this false confidence? Suppose that it would take 10^{50} test cases to completely test a program. (Dick Bender's *Soft Test* program provides estimates of the number of possible test cases of a program. Some programs have yielded estimates as high as 10^{100}.) A regression test suite with 10,000 test cases would be a good sized set, as regression tests go, but of the original 10^{50}, there would still be 9,999,999,999,999,999,999,999,999,999,999,999,999,999,999,999,990,000 more test cases. Any of these could yield a bug that could not be found by the paltry few regression tests that we can run. How can we say that rerunning 10,000 that the program has passed before should give us confidence that it won't fail any of these many others? (*See also* Marick, 1997.)

Rather than running the same tests over and over again, skilled manual testers often run new variations as they retest the program. The new variations address the same areas of the program, but they use different data or order tasks differently or combine tests that were run separately before.

I agree that there is value in being able to conduct smoke tests (see below) or in being able to run a final qualification test at the end of a release. But there are limits to this value. In allocating resources for regression testing (redoing old tests), we should allocate significant resources for tests that have never been run before, i.e. for exploratory or ad hoc testing. Otherwise, we will miss many problems.

5. Don't use the instability of the code as an excuse.

Some people advise testers not to do any automation until the program's user interface has completely stabilized. Projects are said to have failed because testers tried to automate too early. Development groups are said to be bad and uncooperative because they won't follow tidy waterfall methods that would (allegedly) make life easier for testers.

Unfortunately, the GUI changes and changes throughout the development of the product. Taxes get collected. Rains cause floods. User interfaces change. Saying that code isn't ready for automated testing until the UI is frozen seems like saying that code isn't ready for testing until all the bugs have been taken out.

The fact that the UI will change through the life of the product makes it challenging to develop GUI-level automated tests. If you use capture/replay tools or stand-alone scripts that don't hide details of the user interface behind a framework, you will not be able to cope with the UI changes. Change adds complexity that you will have to manage if you automate, but that doesn't give you an excuse to not automate (if you need automation) or to automate badly.

6. Don't put off finding bugs in order to write test cases.

Take seriously the notion that it will take ten times as long to create and minimally document the automated test as it will to design and run it by hand once. If you do extensive test automation early in the project, your automation staff will be running one-tenth as many tests as they would if they were testing manually and you will probably therefore find one-tenth as many bugs. The later you find bugs, the more expensive they are to fix, and the more likely they are to be deferred (not fixed).

If you're going to spend significant time automating tests, you need a staffing strategy that protects your early productivity. This might mean increasing the size of the test team.

7. Don't write simplistic test cases.

One strategy for creating automated tests is to create braindead simple tests that just pull down menus, peek at dialogs, and play one at a time with other elements of the user interface. Early in testing, these are easy to design and the program might not be capable of running more complex test cases. Later, though, these tests are weak, especially in comparison to the increasingly harsh testing done by a skilled manual tester.

If you create a library of 10,000 trivial tests, you'll have executives strutting about the company saying "We have a 10,000 test library. Our testing is superb." But you will have barely scratched the surface of the program.

8. Don't shoot for "100% automation."

Some people recommend that testers automate 100% of their test cases. I strongly disagree with this. I create and run many black box tests only once. Many, many bugs are found during these exploratory tests. To automate these one-shot tests, I would have to spend substantially more time and money per test. In the same period of time, I wouldn't be able to run as many tests. Why should I seek lower coverage at a higher cost per test?

9. Don't use capture/replay to create tests.

Capture utilities can help you script tests by showing you how the test tool interprets a manual test case. They are not useless. But they are dangerous if you try to do too much with them.

In your first course on programming, you probably learned not to write programs like this:

```
SET A = 2
SET B = 3
PRINT A+B
```

Embedding constants in code is obviously foolish. But that's what we do with capture utilities. We create a test script by capturing an exact sequence of exact keystrokes, mouse movements, or commands. These are constants, just like 2 and 3. The slightest change to the program's user interface and the script is invalid. The maintenance costs associated with captured test cases are unacceptable.

10. Don't write isolated scripts in your spare time.

Test groups often try to create automated test cases in their spare time. The overall plan seems to be, "Create as many tests as possible." There is no unifying plan or theme. Each test case is designed and coded independently, and different scripts often repeat exact sequences of commands. For example, dozens of different scripts might all print, and each one would walk through the UI to get to the print dialog.

Test automation is like any other kind of automation. Like automating accounting. You have a bunch of tasks done by hand and you want to replace the hand with the computer. In an automated test application, each test case is a feature.

Imagine writing an accounting program like many people write automated test applications. We'll give it 100,000 independently coded features. Today, we'll run the Add-all-numbers-on-a-ledger-page-with-28-numbers feature. Then we'll run the Add-all-numbers-on-a-ledger-page-with-27-numbers feature. After that, we'll run the Print-two-ledger-pages feature. Each of these features has its own code to read ledger pages. If you change the design of a ledger page, each one has to be revised or, probably, rewritten.

Long ago, programmers discovered subroutines. Modular programming. Structured programming. One of the basic ideas was that you would write one body of code (the subroutine) to do a commonly-done task. All other routines that needed to do that task would call this subroutine. If you automated the task incorrectly, or needed to generalize it or change it, you would change it in this one place and all the other parts of the program would now do the right thing, because they all came here for this task and it got fixed.

No programmer in her right mind would write a significant amount of code without setting commonly used elements into subroutines, modules, procedures, or objects. Why should we do it in testing?

11. Don't create test scripts that won't be easy to maintain over the long term.

Maintenance requirements don't go away just because your friendly automated tool vendor forgot to mention them.

An automated regression test has little power in Release N (the Release for which it was created). However, code changes significantly between Release N and Release N+1. The context around the code changes significantly when you change hardware platforms (Release N for Windows, then for Mac, then for UNIX). If your test cases survive these changes, you can re-use it to find new bugs that weren't in the original Release or visible on the original platform. There is no manual testing or test preparation cost, so you reap the main benefits of automated regression testing in Release N+1 (and later) and on new platforms.

If you don't write code that is easy to maintain when the UI changes, the odds are good that you will carry very little testing code forward from Release N to N+1. You'll pay the price for the code, but you won't get the benefit.

12. Don't make the code machine-specific.

Beware of creating automated tests just to run on your computer at your favored video resolutions or with your favored hardware. Your hardware will change. And other people will want to borrow your test cases.

13. Don't fail to treat this as a genuine programming project.

Automation of software testing is just like all of the other automation efforts that software developers engage in—except that this time, the testers are writing the automation code.

- It is code, even if the programming language is funky.
- Within an application dedicated to testing a program, every test case is a feature.
- From the viewpoint of the automated test application, every aspect of the underlying application (the one you're testing) is data.

To write a successful application, software developers, including testers, must:

- understand the requirements;

- adopt an architecture that allows them to efficiently develop, integrate, and maintain features and data;
- adopt and live with standards. (I don't mean grand schemes like ISO 9000 or CMM. I mean that it makes sense for two programmers working on the same project to use the same naming conventions, the same structure for documenting their modules, the same approach to error handling, etc.. Within any group of programmers, agreements to follow the same rules are agreements on standards);
- be disciplined.

Of all people, testers must realize just how important it is to follow a disciplined approach to software development instead of using quick-and-dirty design and implementation. Without it, we should be prepared to fail as miserably as so many of the applications we have tested.

14. Don't "forget" to document your work.

Documentation of test cases and testing strategy is often treated as an afterthought by automated test developers. Specific suggestions for documentation techniques will have to wait until another paper, but I want to make the point clearly here that you can't maintain code that you don't understand. You can't effectively reuse test cases if you don't know what tests they include. And you can't trust test results if you don't know how to tell whether the program passed or failed the test.

Without adequate documentation, how can you reap the benefit in Release N+1 of all the automation programming that you did in Release N?

15. Don't deal unthinkingly with ancestral code.

At the start of Release N+1, you might be given a huge set of test cases from Release N. Naturally, these are undocumented or the individual routines are documented (header file by header file) but only in terms of how they work, not in terms of their place in a larger strategy. The files are handed to you reverently—this is code that comes to us from dear departed ancestors (now working for your competition).

How do you know how good this code is? How do you know whether the program has passed or failed each test? How do you know whether all of the tests get executed? How do you know what aspects of the code are covered by these tests and how thoroughly? How do you know what other testing (manual or automated) that you'll have to do to supplement the tests in this set? How much time and effort should you devote to being a software archaeologist, with the goal of unearthing answers to these questions?

I don't have a "correct" answer for this. It is easy to burn off so much time reverse engineering bad code that you lose the opportunity to make it better. It is also common for arrogant programmers to adopt a not-coded-here attitude and throw away superb work. Here are a few suggestions:

- The more stable the last version was in the field, the better tested it probably was, and the more valuable the test set probably is.
- If you are set up to monitor code coverage, run the full set of automated tests with the coverage monitor on. What do you get? One participant at LAWST reported that a large set of test cases yielded only 4% coverage. He spent relatively little further time with them.
- The main value of ancestral code might be the education it gives you about the last testing group's approach to breaking the product. Rather than asking how to get these routines working again, it might be more valuable to ask why these tests were created.

16. Don't give the high-skill work to outsiders.

Many testers are relatively junior programmers. They don't have experience designing systems. To automate successfully, you may have to add more senior programmers to the testing group. Be cautious about using contractors to implement automation. The contractors go away, taking the knowledge with them. And the in-house tester/programmers feel slighted. They feel as though their careers are not being given a chance to develop because the hot new technologies get farmed out to contractors.

Develop skills in-house. Use contractors as trainers or to do the more routine work.

17. Don't insist that all of your testers be programmers.

Many excellent black box testers have no programming experience. They provide subject matter expertise or other experience with customer or communications issues that most programmers can't provide. They are indispensable to a strong testing effort. But you can't expect these people to write automation code. Therefore, you need a staffing and development strategy that doesn't require everyone to write test code. You also want to avoid creating a pecking order that places tester-programmers above tester-non-programmers. This is a common, and in my view irrational and anti-productive, bias in test groups that use automation tools. It will drive out your senior non-programming testers, and it will cost you much of your ability to test the program against actual customer requirements.

18. Don't put up with bugs and crappy support for the test tool.

Test tool vendors have blessed us with the opportunity to appreciate how our customers feel when they try to use software to get real work done. Like everyone else's, their software has bugs and some (maybe most) of them are probably known, deferred bugs—they decided that you wouldn't mind these bugs. They have their own interesting theories about what makes good user interface design, about how people read user documentation, and about the extent to which different problems entitle you to call for support.

Order a copy of the test tool on a trial basis. If you can't get it that way from the manufacturer, order it from a mail order shop that will give you a 60 or 90 day money back guarantee. Try to use the trial period for serious work. When you get stumped or you run into bugs, call the tool vendor for support. If a bug is blocking your testing progress (or looks like it will block you in the future), then you need a fix or a workaround. Otherwise, you'll probably give up on the tool later anyway. If its bugs limit your application development efforts, look for a different tool that has bugs that don't interfere with you. If its support organization is unhelpful (but plenty surly or arrogant), find a vendor that will take care of you.

Don't assume that problems will go away once you understand the tool better.

19. Don't forget to clear up the fantasies that have been spoonfed to your management.

Like our vendor at the start of this section, some vendors' salespeople will make fantastic claims that only a senior executive could believe. You have to reset expectations or you will fail even if you do superb technical work.

Twenty-Seven Questions About Requirements

I follow Brian Lawrence's definition of requirements (Gause & Lawrence, *in preparation*) as "Anything that drives design choices." Several factors determine the probable scope, focus and strategy of a test automation project. The sidebar lists 27 such factors. I hope that these are largely self-explanatory—there just isn't room in this space-limited paper to explore all of them. I will briefly examine a few (item numbers match the numbers in the sidebar):

1. Will the user interface of the application be stable or not?

If the UI will be unstable, maybe you will do only the quick-payback automation tasks (such as device compatibility testing). Or maybe you'll isolate the variability by referring to every feature of the program indirectly, through calls to feature-wrapper functions in an automation framework. Or maybe you'll go below that, calling features through an application programmer's interface (API). Instability is a constraint that complicates life and drives design choices, but it is not a barrier.

2. To what extent are oracles available?

An oracle is a program that tells you what the result of a test case *should* be. Oracles are the critical enablers of high-powered automation that is based on random creation of test cases. With an oracle, you can create massive numbers of tests on the fly and know for each one whether the program has passed or failed the test. (For an example, see the sixth architectural example, on equivalence checking, below.)

3. To what extent are you looking for delayed-fuse bugs (memory leaks, wild pointers, etc.)?

Tests for these often involve long series of random calls to suspect functions in different orders. Sometimes you find the bug just by calling the same function many times. Sometimes the error is triggered only when you call the

27 QUESTIONS ABOUT REQUIREMENTS

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?

right three functions in the right order.

This type of testing often involves connected machines (one serving as master, the other as slave) using the Ferret or Elverex Evaluator (if it's still around) or a comparable test tool that makes the Master machine look to the Slave like a keyboard (for input) and a video monitor (for output). The Master drives the slave through different sequences of tasks, tracks what tasks were performed in what order, and tracks state information sent to it by the Slave, such as remaining amount of available memory after each task. When the Slave crashes, the Master still has all of this data.

7. Do you expect this product to sell through multiple versions?

The series of questions starting with this one ask how maintainable your test suite has to be. If you will only sell one version of the program (such as a computer game), in one language, with few or no expected bug-fix updates, then you cannot count on cost savings from testing later versions more efficiently to pay for the expense of automation during the development of this product.

11. What level of source control has been applied to the code under test?

If there is poor source control then a bug that was fixed last week might come back next week, just because someone is doing maintenance using an out-of-date version of the code. The poorer the source control, the more essential it is to do regression testing.

27 QUESTIONS ABOUT REQUIREMENTS

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?

19. Is the quality of your product driven primarily by regulatory or liability considerations or by market forces (competition)?

The series of questions starting with this one raise the point that sometimes there are important reasons for automating and documenting test cases that have nothing to do with efficiency of finding bugs. If you have a need to be able to produce proof that you ran certain types of tests, having the tests themselves, ready to run, is valuable documentation.

Planning for Short-Term and Long-Term Returns on Investment

Most of the time, GUI-level regression tests cost enough to create that they don't pay for themselves in the release in which they were created. You will start enjoying the benefits of a test created in Release N during the testing of Release N.01 or N+1.

There are ways to realize short-term gains from test automation. Here are some examples.

- There's a big payoff in automating a suite of acceptance-into-testing (also called "smoke") tests. You might run these 50 or 100 times during development of Release N. Even if it takes 10x as long to develop each test as to execute each test by hand, and another 10x cost for maintenance, this still creates a time saving equivalent to 30-80 manual executions of each test case.
- You can save time, reduce human error, and obtain good tracking of what was done by automating configuration and compatibility testing. In these cases, you are running the same tests against many devices or under many environments. If you test the program's compatibility with 30 printers, you might recover the cost of automating this test in less than a week.
- Many stress tests require more activity than you can do manually (such as taking input from 100 terminals at the same time.) Or they require repeated executions of a set of commands in order to create huge data files. Automating tasks like these makes it possible for you to do tests that you could not otherwise do. There is no comparison to the amount of time it would take to do the test manually, because you wouldn't do it manually. If you want to run this type of test, you automate it.
- Regression automation facilitates performance benchmarking across operating systems and across different development versions of the same program.

Take advantage of opportunities for near-term payback from automation, but be cautious when automating with the goal of short-term gains. Cost-justify each additional test case, or group of test cases.

If you are looking for longer term gains, then along with developing quick-payback tests for Release N, you you should be developing scaffolding that will make for broader and more efficient automated testing in Version N+1. The discussion of frameworks (below) illustrates a type of scaffolding that I have in mind.

Six Successful Architectures

The architecture of a software product specifies the overall technical strategy for the product's development. For example, the architecture:

- provides a strategy for subdividing the overall product into a set of components.
- determines the means by which one component can pass data or a request for service to another.

- specifies the relationship between code and data. (As you'll see in the discussion of the data-driven architecture, some elements of a product can be represented either way.)
- determines the method by which the program will respond to input in real time (e.g. adoption of a polling strategy to check for device inputs or of an event driven strategy).

Your decisions about the architecture of the software (in this case, the test automation application) are fundamental to the rest of your design and implementation of the program. These decisions should be guided by your requirements. Different requirements call for different solutions, which call for different architectures.

SIX SUCCESSFUL ARCHITECTURES

1. The quick and dirty architecture.
2. The framework-based architecture.
3. The data-driven architecture.
4. Application-independent data-driven testing.
5. Real-time simulator with event logs.
6. Equivalence testing: Random testing with an oracle.

Here are a few examples of architectures that you could use. Note that this is not close to being an exhaustive list. For example, there is nothing specially designed to take advantage of an object-oriented architecture of the underlying program being tested. Additionally, the real-time simulator is the only architecture that addresses event-driven programming in the underlying program.

1. The Quick and Dirty Architecture

If you're looking for fast payback from a narrow range of tests, it probably won't be worth it to build a framework or a data-driven model.

Imagine doing printer compatibility testing, for example. Perhaps you'll test 40 printers during a compatibility run. Suppose there are 10 tests in the suite.You can either run the tests on each printer by hand or you can create an automated test series. Even if you'll only use these tests this one week, it will be more cost-effective to automate than to run these many tests by hand.

What architecture is appropriate for these tests? *Whatever works.* It would be nice to put these tests inside a framework or a data driver but the tests will have paid for themselves even if they are completely unmaintainable.

Fast-payback tests are good candidates for use when your group is just learning how to use a new testing tool. You can afford to write throwaway code because you'll still improve productivity over manual testing. Later, when you know the tool better, you can do a better job with the successors to these tests and to more subtle ones.

2. The framework-based architecture.

The *framework* is often used in conjunction with one or more data-driven testing strategies. Tom Arnold (1996) discusses this approach well. Hayes (1995) and Groder (1997) give helpful descriptions that are complementary to Arnold's.

The framework isolates the application under test from the test scripts by providing a set of functions in a shared function library. The test script writers treat these functions as if they were basic commands of the

test tool's programming language. They can thus program the scripts independently of the user interface of the software.

For example, a framework writer might create the function, `openfile(p)`. This function opens file p. It might operate by pulling down the file menu, selecting the Open command, copying the file name to the file name field, and selecting the OK button to close the dialog and do the operation. Or the function might be richer than this, adding extensive error handling. The function might check whether file p was actually opened or it might log the attempt to open the file, and log the result. The function might pull up the File Open dialog by using a command shortcut instead of navigating through the menu. If the program that you're testing comes with an application programmer interface (API) or a macro language, perhaps the function can call a single command and send it the file name and path as parameters. The function's definition might change from week to week. The scriptwriter doesn't care, as long as `openfile(x)` opens file x.

Many functions in your library will be useful in several applications (or they will be if you design them to be portable). Don't expect 100% portability. For example, one version of `openfile()` might work for every application that uses the standard File Open dialog but you may need additional versions for programs that customize the dialog.

Frameworks include several types of functions, from very simple wrappers around simple application or tool functions to very complex scripts that handle an integrated task. Here are some of the basic types:

2a. Define every feature of the application.

You can write functions to select a menu choice, pull up a dialog, set a value for a variable, or issue a command. If the UI changes how one of these works, you change how the function works. Any script that was written using this function changes automatically when you recompile or relink.

Frameworks are essential when dealing with custom controls, such as *owner-draw controls*. An owner-draw control uses programmer-supplied graphics commands to draw a dialog. The test-automation tool will know that there is a window here, but it won't know what's inside. How do you use the tool to press a button in a dialog when it doesn't know that the button is there? How do you use the tool to select an item from a listbox, when it doesn't know the listbox is there? Maybe you can use some trick to select the third item in a list, but how do you select an item that might appear in any position in a variable-length list? Next problem: how do you deal consistently with these invisible buttons and listboxes and other UI elements when you change video resolution?

At the first LAWST meeting, we talked of kludges upon kludges to deal with issues like these. Some participants estimated that they spent half of their automation development time working around the problems created by custom controls.

These kludges are a complex, high-maintenance, aggravating set of distractions for the script writer. I call them distractions because they are problems with the tool, not with the underlying program that you are testing. They focus the tester on the weaknesses of the tool, rather than on finding and reporting the weaknesses of the underlying program.

If you must contend with owner-draw controls, encapsulating every feature of the application is probably your most urgent large task in building a framework. This hides each kludge inside a function. To use a feature, the programmer calls the feature, without thinking about the kludge. If the UI changes, the kludge can be redone without affecting a single script.

2b. Define commands or features of the tool's programming language.

The automation tool comes with a scripting language. You might find it surprisingly handy to add a layer of indirection by putting a wrapper around each command. A wrapper is a routine that is created around another function. It is very simple, probably doing nothing more than calling the wrapped function. You

can modify a wrapper to add or replace functionality, to avoid a bug in the test tool, or to take advantage of an update to the scripting language.

Tom Arnold (1996) gives the example of `wMenuSelect`, a Visual Test function that selects a menu. He writes a wrapper function, `SelMenu()` that simply calls `wMenuSelect`. This provides flexibility. For example, you can modify `SelMenu()` by adding a logging function or an error handler or a call to a memory monitor or whatever you want. When you do this, every script gains this new capability without the need for additional coding. This can be very useful for stress testing, test execution analysis, bug analysis and reporting and debugging purposes.

LAWST participants who had used this approach said that it had repeatedly paid for itself.

2c. Define small, conceptually unified tasks that are done frequently.

The `openfile()` function is an example of this type of function. The scriptwriter will write hundreds of scripts that require the opening of a file, but will only consciously care about how the file is being opened in a few of those scripts. For the rest, she just wants the file opened in a fast, reliable way so that she can get on with the real goal of her test. Adding a library function to do this will save the scriptwriter time, and improve the maintainability of the scripts.

This is straightforward code re-use, which is just as desirable in test automation as in any other software development.

2d. Define larger, complex chunks of test cases that are used in several test cases.

It may be desirable to encapsulate larger sequences of commands. However, there are risks in this, especially if you overdo it. A very complex sequence probably won't be re-used in many test scripts, so it might not be worth the labor required to generalize it, document it, and insert the error-checking code into it that you would expect of a competently written library function. Also, the more complex the sequence, the more likely it is to need maintenance when the UI changes. A group of rarely-used complex commands might dominate your library's maintenance costs.

2e. Define utility functions.

For example, you might create a function that logs test results to disk in a standardized way. You might create a coding standard that says that every test case ends with a call to this function.

Other examples of members of this class, functions to synchronize the application and the data (including functions to restore the application to its starting state or to some known state), functions to enable error recovery (including restarting the underlying application if necessary) and functions to report on the state of the system (such as available memory and resources). Hayes (1995) and Groder (1997) provide several useful descriptions and explanations of these and other important utilities.

Each of the tools provides its own set of pre-built utility functions. You might or might not need many additional functions.

Some framework risks

You can't build all of these commands into your library at the same time. You don't have a big enough staff. Automation projects have failed miserably because the testing staff tried to create the ultimate, gotta-have-everything programming library. Management support (and some people's jobs) ran out before the framework was completed and useful. You have to prioritize. You have to build your library over time.

Don't assume that everyone will use the function library just because it's there. Some people code in different styles from each other. If you don't have programming standards that cover variable naming, order of parameters in function interfaces, use of global variables, etc., then what seems reasonable to one person will seem unacceptable to another. Also, some people hate to use code they didn't write. Others

come onto a project late and don't know what's in the library. Working in a big rush, they start programming without spending any time with the library. You have to manage the use of the library.

Finally, be careful about setting expectations, especially if your programmers write their own custom controls. In Release 1.0 (or in the first release that you start automating tests), you will probably spend most of your available time creating a framework that encapsulates all the crazy workarounds that you have to write just to press buttons, select list items, select tabs, and so on. The payoff from this work will show up in scripts that you finally have time to write in Release 2.0. Framework creation is expensive. Set realistic expectations or update your resume.

3. The data-driven architecture.

The other approach (besides frameworks) most explored at the first LAWST was data driven. Data-driven design and framework-based design can be followed independently or they can work well together as an integrated approach.

The following example comes out of my work experience (in 1993). This is a straightforward application of a common programming style. Buwalda (1996), Hayes (1995), and Pettichord (1995) have published interesting variations on this theme.

Imagine testing a program that lets the user create and print tables. Here are some of the things you can manipulate:

- The table caption. It can vary in typeface, size, and style (italics, bold, small caps, or normal).
- The caption location (above, below, or beside the table) and orientation (letters are horizontal or vertical).
- A caption graphic (above, below, beside the caption), and graphic size (large, medium, small). It can be a bitmap (PCX, BMP, TIFF) or a vector graphic (CGM, WMF).
- The thickness of the lines of the table's bounding box.
- The number and sizes of the table's rows and columns.
- The typeface, size, and style of text in each cell. The size, placement, and rotation of graphics in each cell.
- The paper size and orientation of the printout of the table.

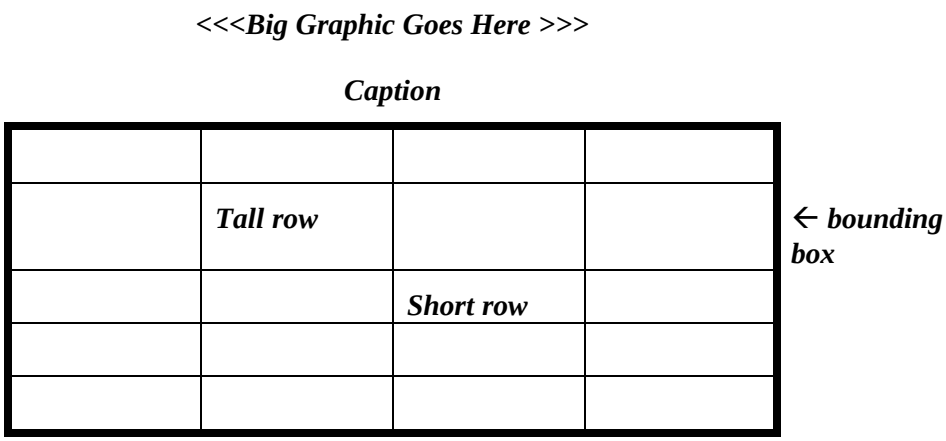

Figure 1 Some characteristics of a table

These parameters are related because they operate on the same page at the same time. If the rows are too big, there's no room for the graphic. If there are too many typefaces, the program might run out of memory. This example cries out for testing the variables in combination, but there are millions of combinations.

Imagine writing 100 scripts to test a mere 100 of these combinations. If one element of the interface should change—for example, if Caption Typeface moves from one dialog box to another—then you might have to revise each script.

	Caption location	Caption typeface	Caption style	Caption Graphic (CG	CG format	CG size	Bounding box width
1	Top	Times	Normal	Yes	PCX	Large	3 pt.
2	Right	Arial	Italic	No			2 pt.
3	Left	Courier	Bold	No			1 pt.
4	Bottom	Helvetica	Bold Italic	Yes	TIFF	Medium	none

Figure 2 The first few rows of a test matrix for a table formatter

Now imagine working from a test matrix. A test case is specified by a combination of the values of the many parameters. In the matrix, each row specifies a test case and each column is a parameter setting. For example, Column 1 might specify the Caption Location, Column 2 the Caption Typeface, and Column 3 the Caption Style. There are a few dozen columns.

Create your matrix using a spreadsheet, such as Excel.

To execute these test cases, write a script that reads the spreadsheet, one row (test case) at a time, and executes mini-scripts to set each parameter as specified in the spreadsheet. Suppose that we're working on Row 2 in Figure 2's matrix. The first mini-script would read the value in the first column (Caption Location), navigate to the appropriate dialog box and entry field, and set the Caption Location to *Right*, the value specified in the matrix . Once all the parameters have been set, you do the rest of the test. In this case, you would print the table and evaluate the output.

The test program will execute the same mini-scripts for each row.

In other words, your structure is:

```
Load the test case matrix
For each row I                                  (row = test case)
     For each column J                          (column = parameter)
          Execute the script for this parameter:
               -  Navigate to the appropriate dialog
                  box or menu
               -  Set the variable to the value
                  specified in test item (I,J)
```

```
Run test I and evaluate the results
```

If the program's design changed, and the Caption Location was moved to a different dialog, you'd only have to change a few lines of code, in the one mini-script that handles the caption location. You would only have to change these lines once: this change will carry over to every test case in the spreadsheet. This separation of code from data is tremendously efficient compared to modifying the script for each test case.

There are several other ways for setting up a data-driven approach. For example, Bret Pettichord (1996) fills his spreadsheet with lists of commands. Each row lists the sequence of commands required to execute a test (one cell per command). If the user interface changes in a way that changes a command sequence, the tester can fix the affected test cases by modifying the spreadsheet rather than by rewriting code. Other testers use sequences of simple test cases or of machine states.

Another way to drive testing with data uses previously created documents. Imagine testing a word processor by feeding it a thousand documents. For each document, the script makes the word processor load the document and perform a sequence of simple actions (such as printing).

A well-designed data-driven approach can make it easier for non-programming test planners to specify their test cases because they can simply write them into the matrix. Another by-product of this approach, if you do it well, is a set of tables that concisely show what test cases are being run by the automation tool.

It is natural, if not essential, for the miniscripts that execute each column to be maintained inside a testing framework.

4. Application-independent data-driven testing

The next figure is a test matrix that Hung Nguyen and I developed. Matrices like this are handy for capturing the essence of some repetitive types of testing. The columns represent the types of tests that you can run. (If you are doing manual testing, you will probably sample from these rather than running each test for each variable under test.) The rows are for the objects you are testing.

In this case, we are testing numeric input fields. Imagine testing a new program. You would scan through all of the dialog boxes (or other data entry opportunities in the software) and list the numeric input fields. Through trial and error (or from the specification if one exists and provides this data), you also note beside the field the range of valid values. For example, the first column in one row might read "Print: # of copies (1-99)." This shows the dialog name (Print), the field name (# of copies), and the range of values (1-99). For each such field, you perform some or all of the tests listed in the columns, filling in the cells with results. (I use highlighters, coloring a cell green if the program passed and pink if it failed a given test.)

Elizabeth Hendrickson (personal communication, with some follow-up at the third LAWST) suggested that ths type of matrix could be automated. For each field, all you would need would be an access method (a short script) to get to the place where you enter data, plus the high and low bounds, plus the text of error messages that the program will generate for too-big values, too-small values, and otherwise-inappropriate (e.g. non-numeric) values. This would be relatively easy to set up for any program and so you have a package of common tests that can be carried from one application that you're testing to another.

This approach differs strongly from regression testing because you might use this tool to execute these tests for the first time in the program that you're testing. You skip the manual execution phase completely (or do just enough to reverse engineer the upper and lower bounds and the error messages). Rather than slowing you down your testing while you automate, this should speed you up by racing through standard situations very quickly, very early in testing.

Test Matrix for a Numeric Input Field

Additional Instructions:

		Nothing	Valid value	At LB of value	At UB of value	At LB of value - 1	At UB of value + 1	Outside of LB of value	Outside of UB of value	0	Negative	At LB number of digits or chars	At UB number of digits or chars	Empty field (clear the default value)	Outside of UB number of digits or chars	Non-digits	Wrong data type (e.g. decimal into integer)	Expressions	Space	Non-printing char (e.g., Ctrl+char)	DOS filename reserved chars (e.g., "\ * . :")	Upper ASCII (128-254)	Upper case chars	Lower case chars	Modifiers (e.g., Ctrl, Alt, Shift-Ctrl, etc.)	Function key (F2, F3, F4, etc.)

Hendrickson has been sketching out prototype code to do this kind of work. I hope that she'll publish her results some day.

This example illustrates a broader idea. We have common test cases across many applications. You have to test push buttons (to see if they pushed), menu displays, input fields, and so on. Rather than creating tests for these mundane issues time and time again, we should be able to create standard routines that port quickly from program-under-test to program-under-test.

5. Real-time simulator with event logs

Imagine working with a multi-user, multi-tasking, event-driven program. For example, imagine trying to test the software in a digital PBX (private telephone system). Any number of events can happen at any time. How do you test this? (Note: this approach and its cousin, smart monkeys, have been briefly noted but not yet thoroughly discussed in the LAWST meetings. These notes are my own.)

The simulator is programmed to be able to generate any (or many) of the events that can arise in the system. It can initiate a new phone call, hang up, place a call on hold, reconnect a held call, initiate a conference call, send a fatal exception message that will cause the server to redistribute the calls associated with this phone while the (digital, multi-feature) phone reboots and reloads its software, and so on. The simulator can emulate many phones at once and generate many events almost simultaneously. In effect, the simulator is playing the role of many clients in a client/server system. (Or you can run many instances of a simulator on each of many client machines.)

On the server side, one process generates event logs. These dump debug messages to a hard copy device (handy in case the server crashes and loses its data) or to a disk file or to a remote machine that won't crash when the server goes down. The contents of the event logs are error messages, warning messages and other diagnostics written into the code by the programmers whenever they run into an unusual event or a common event arising under suspicious circumstances. For example, a programmer might send a message to the log when a function is called with a valid but rare value in one of its parameters.

The event logger gives running information on the stability of the system and is normally useful for maintenance staff. However, in conjunction with a simulator, it is tremendously useful as a support for exploratory testing.

Imagine going to the programmers and saying that you'll be testing the Caller ID and Caller ID Blocking features for the next few days. They insert a collection of new diagnostics into their code, to flag unusual events associated with these features. You run the simulator overnight and get a printout with lots of diagnostic messages. Many of these will look harmless and will in fact reflect normal performance. (You may have to chat with the programmers who wrote the messages to discover this.) Others will look more suspicious. To track down the cause of these, the programmer might suggest lines of manual testing for you to try out this afternoon. You test with the same software build and see if you can get the same patterns of log output. Based on your results, you might discover that the system is functioning normally and reasonably, or that you have discovered a bug, or that more investigation is needed. In the latter case, the programmer revises the debug messages and the new results show up in the next set of logs from the next simulation.

The toughest challenges of this approach are to keep the volume of messages flowing to the logs within reason and to make the diagnostic messages concise but recognizable and interpretable. Messages that report normal performance must be stripped out, especially if they are printed frequently. Or there should be two logs, one a "full" log and one a log that only shows debug events that have been classed as being of a more serious nature than information and state messages. Additionally, when you switch focus from one cluster of features to another it is important to clean out the extensive set of messages that were put in to

support your testing of the first cluster of features. Otherwise you won't easily be able to notice what is new.

I stressed the event-driven nature of the application under test because this type of simulator seems especially well-suited to aid in exploratory testing in complex, event-driven situations. However, it also makes sense for applications whose style is more procedural (or single user, single task) when the order of feature execution is an issue. The random switching between functions/features gives the program a chance to fail in the face of complex combinations of functions/features. The event log (such as the Windows debug monitor) reports back unusual events as they are detected. The style of follow-up is the same.

The beauty of this approach is that it is such a support for exploratory testing. It drives the software under circumstances that approach real use. Its random number generator keeps it constantly generating fresh test cases and combinations of test cases. The log reports actual software failures and it also reports that some things bear further investigation. In a complex product with many interactions, this is an invaluable supplement to the error guessing that experienced testers do so well with systems or circumstances that they understand.

The simulators that I've seen have been custom-programmed tools. However, even though GUI test tools were not explicitly designed to be simulators in this way, I don't see any reason that you can't or shouldn't use them this way.

6. Equivalence testing: Random testing with an oracle

Kaner, Falk & Nguyen (1993) describe this approach as function equivalence testing. Here's an example of classic function equivalence testing.

Suppose you were developing a spreadsheet program that would have the same functions as Excel 95. Suppose too that you were satisfied with the computational accuracy of Excel 95. Then you could set up a random test architecture in which you randomly select functions (such as the logarithm function), operators (plus, minus, etc.), and values (such as 5, in log(5)) to create random expressions. Evaluate the expression in your program, evaluate the same expression in Excel, and compare the results. If they match within round-off error, your program passes the test. If not, you've found a bug.

Generalizing this, you are using Excel 95 as an oracle. You have a program whose functionality is so similar to the oracle's that you can usefully compare results.

The same principles could apply if you test a database search routine by using two database management programs (yours and a respected competitor's or a prior version of your own) to search through the same underlying set of data. You could also do this to check (across communication programs) dialing strings and data transmissions through modems. You might do this to check output of identical pictures to printers (but it won't work if the printer randomly changes printout bitmaps in the process of dithering).

The point in each case is the same. For a feature or group of features in your program, find a comparable feature (or group) in someone else's program. Use your GUI regression test tool (with a random number generator) to generate test data to feed to each program and compare the results.

Note that you are not doing regression testing. All of these tests are new. The program has a reasonable opportunity to fail them. (That is, it hasn't already proved that it didn't fail them before.) No individual test might be as powerful as a well crafted special case (e.g. a combination of many boundary cases), but thousands or millions of tests are run and collectively their power to find bugs is impressive.

Conclusions Reached in the Los Altos Workshop on Software Testing

This section is taken directly from Kaner (1997). It is a powerful set of materials to show to your management if they are being misinformed by a vendor, by other testers, by magazine articles, or by other executives who don't know much about testing.

At the LAWST meeting, during the last third of each day, we copied several statements made during the discussion onto whiteboards and voted on them. We didn't attempt to reach consensus. The goal was to gauge the degree to which each statement matched the experience of several experienced testers. In some cases, some of us chose not to vote, either because we lacked the specific experience relevant to this vote, or because we considered the statement ill-framed. (I've skipped most of those statements.)

If you're trying to educate an executive into costs and risks of automation, these vote tallies might be useful data for your discussions.

General Principles

1. These statements are not ultimate truths. In automation planning, as in so many other endeavors, you must keep in mind what problem are you trying to solve, and what context are you trying to solve it in. *(Consensus)*
2. GUI test automation is a significant software development effort that requires architecture, standards, and discipline. The general principles that apply to software design and implementation apply to automation design and implementation. *(Consensus)*
3. For efficiency and maintainability, we need first to develop an automation structure that is invariant across feature changes; we should develop GUI-based automation content only as features stabilize. *(Consensus)*
4. Several of us had a sense of patterns of evolution of a company's automation efforts over time:

   First generalization *(7 yes, 1 no):* In the absence of previous automation experience, most automation efforts evolve through:

   a) Failure in capture /playback. It doesn't matter whether we're capturing bits or widgets (object oriented capture/replay);

   b) Failure in using individually programmed test cases. (Individuals code test cases on their own, without following common standards and without building shared libraries.)

   c) Development of libraries that are maintained on an ongoing basis. The libraries might contain scripted test cases or data-driven tests.

   Second generalization *(10 yes, 1 no):* Common automation initiatives failures are due to:

   a) Using capture/playback as the principle means of creating test cases;

   b) Using individually scripted tested cases (i.e. test cases that individuals code on their own, without following common standards and without building shared libraries);

   c) Using poorly designed frameworks. This is a common problem.

5. Straight replay of test cases yields a low percentage of defects. *(Consensus)*

   Once the program passes a test, it is unlikely to fail that test again in the future. This led to several statements (none cleanly voted on) that automated testing can be dangerous because it can gives us a falsely warm and fuzzy feeling that the program is not broken. Even if the program isn't broken today in the ways that it wasn't broken yesterday, there are probably many ways in which the program is broken. But you won't find them if you keep looking where the bugs aren't.

6. Of the bugs found during an automated testing effort, 60%-80% are found during development of the tests. That is, unless you create and run new test cases under the automation tool right from the start, most bugs are found during manual testing. *(Consensus)*

   (Most of us do not usually use the automation tool to run test cases the first time. In the traditional paradigm, you run the test case manually first, then add it to the automation suite after the program passes the test. However, you *can* use the tool more efficiently if you have a way of determining whether the program passed or failed the test that doesn't depend on previously captured output. For example:

   - Run the same series of tests on the program across different operating system versions or configurations. You may have never tested the program under this particular environment, but you know how it should work.
   - Run a function equivalence test (*see* Kaner, et al., 1993). In this case, you run two programs in parallel and feed the same inputs to both. The program that you are testing passes the test if its results always match those of the comparison program.
   - Instrument the code under test so that it will generate a log entry any time that the program reaches an unexpected state, makes an unexpected state transition, manages memory, stack space, or other resources in an unexpected way, or does anything else that is an indicator of one of the types of errors under investigation. Use the test tool to randomly drive the program through a huge number of state transitions, logging the commands that it executes as it goes. The next day, the tester and the programmer trace through the log looking for bugs and the circumstances that triggered them. This is a simple example of a simulation. If you are working in collaboration with the application programming team, you can create tests like this that might use your tool more extensively and more effectively (in terms of finding new bugs per week) than you can achieve on your own, scripting new test cases by hand.)

7. Automation can be much more successful when we collaborate with the programmers to develop hooks, interfaces, and debug output. *(Consensus)*

   Many of these collaborative approaches don't rely on GUI-based automation tools, or they use these tools simply as convenient test drivers, without regard to what I've been calling the basic GUI regression paradigm. It was fascinating going around the table on the first day of LAWST, hearing automation success stories. In most cases, the most dramatic successes involved collaboration with the programming team, and didn't involve traditional uses (if any use) of the GUI-based regression tools.

   We will probably explore collaborative test design and development in a later meeting of LAWST.

8. Most code that is generated by a capture utility is unmaintainable and of no long term value. However, the capture utility can be useful when writing a test because it shows how the tool interprets a series of recent events. The script created by the capture tool can give you useful ideas for writing your own code. *(Consensus)*

9. We don't use screen shots "at all" because they are a waste of time. (Actually, we mean that we hate using screen shots and use them only when necessary. We do find value in comparing small sections of the screen. And sometimes we have to compare screen shots, perhaps because we're testing an owner-draw control. But to the extent possible, we should be comparing logical results, not bitmaps.) *(Consensus)*

10. Don't lose site of the testing in test automation. It is too easy to get trapped in writing scripts instead of looking for bugs. *(Consensus)*

Test Design

11. Automating the easy stuff is probably not the right strategy. *(Consensus)*

    If you start by creating a bunch of simple test cases, you will probably run out of time before you create the powerful test cases. A large collection of simple, easy-to-pass test cases might look more rigorous than ad hoc manual testing, but a competent manual tester is probably running increasingly complex tests as the program stabilizes.

12. Combining tests can find new bugs (the sum is greater than the parts). *(Consensus)*

13. There is value in using automated tests that are indeterminate (i.e. random) though we need methods to make a test case determinate. *(Consensus)*

    We aren't advocating blind testing. You need to know what test you've run. And sometimes you need to be able to specify exact inputs or sequences of inputs. But if you can determine whether or not the program is passing the tests that you're running, there is a lot to be said for constantly giving it new test cases instead of reruns of old tests that it has passed.

14. We need to plan for the ability to log what testing was done. *(Consensus)*

    Some tools make it easier to log the progress of testing, some make it harder. For debugging purposes and for tracing the progress of testing, you want to know at a glance what tests cases have been run and what the results were.

Staffing and Management

15. Most of the benefit from automation work that is done during Release N (such as Release 3.0) is realized in Release N+1. There are exceptions to this truism, situations in which you can achieve near-term payback for the automation effort. Examples include smoke tests, some stress tests (some stress tests are impossible *unless* you automate), and configuration/compatibility tests. *(Consensus)*

16. If Release N is the first release of a program that you are automating, then your primary goal in Release N may be to provide scaffolding for automation to be written in Release N+1. Your secondary goal would be light but targeted testing in N. *(Consensus)*

17. People need to *focus* on automation not to do it as an on-the-side task. If no one is *dedicated* to the task then the automation effort is probably going to be a waste of time. *(Consensus)*

18. Many testers are junior programmers who don't know how to architect or create well designed frameworks. *(Consensus)*

Data-Driven Approach

The data-driven approach was described in the main paper. I think that it's safe to say that we all like data-driven approaches, but that none of us would use a data-driven approach in every conceivable situation. Here are a few additional, specific notes from the meeting.

19. The subject matter (the data) of a data-driven automation strategy might include (for example):

    - parameters that you can input to the program;
    - sequences of operations or commands that you make the program execute;
    - sequences of test cases that you drive the program through;
    - sequences of machine states that you drive the program through;
    - documents that you have the program read and operate on;

- parameters or events that are specified by a model of the system (such as a state model or a cause-effect-graph based model) *(Consensus).*

20. Data-driven approaches can be highly maintainable and can be easier for non-programmers to work with. *(Consensus)*

    Even though we all agreed on this in principle, we had examples of disorganized or poorly thought out test matrices, etc. If you do a poor job of design, no one will be able to understand or maintain what you've done.

21. There can be multiple interfaces to enter data into a data file that drives data-driven testing. You might pick one, or you might provide different interfaces for testers with different needs and skill sets. *(Consensus)*

Framework-Driven Approach

For a while, the framework discussion turned into an extended discussion of the design of a procedural language, and of good implementation practices when using a procedural language. We grew tired of this, felt that other people had tackled this class of problem before, and we edited out most of this discussion from our agreements list. I've skipped most of the remaining points along these lines. Here are a few framework-specific suggestionss:

22. The degree to which you can develop a framework depends on the size / sophistication of your staff. *(Consensus).*

23. When you are creating a framework, be conscious of what level you are creating functions at. For example, you could think in terms of operating at one of three levels:

    - menu/command level, executing simple commands;
    - object level, performing actions on specific things;
    - task level, taking care of specific, commonly-repeated tasks.

    You might find it productive to work primarily at one level, adding test cases for other levels only when you clearly need them.

    There are plenty of other ways to define and split levels. Analyze the task in whatever way is appropriate. The issue here is that you want to avoid randomly shifting from creating very simple tests to remarkably long, complex ones. *(Consensus)*

24. Scripts loaded into the framework's function library should generally contain error checking. *(Consensus)*

    This is good practice for any type of programming, but it is particularly important for test code because we expect the program that we're testing to be broken, and we want to see the first symptoms of a failure in order to make reporting and troubleshooting easier.

25. When creating shared library commands, there are risks in dealing with people's differing programming and documentation styles. People will not use someone else's code if they are not comfortable with it.*(Consensus)*

26. Beware of "saving time" when you create a scripts by circumventing your library. Similarly, beware of not creating a library. *(Consensus)*

    The library is an organized repository for shared functions. If a function is too general, and requires the user to pass it a huge number of parameters, some programmers (testers who are doing automation) will prefer to use their own special-purpose shorter version. Some programmers in a hurry simply

won't bother checking what's in the library. Some programmers won't trust the code in the library because they think (perhaps correctly) that most of it is untested and buggy.

27. It is desirable to include test parameters in data files such as .ini files, settings files, and configuration files rather than as constants embedded into the automation script or into the file that contains the script. *(Consensus)*
28. Wrappers are a good thing. Use them as often as reasonably possible. *(Consensus)*

Localization

We spent a lot of time talking about localization, and we came to conclusions that I found surprising. We'll probably revisit these in later LAWST meetings, but the frustration expressed in the meeting by people who had automated localization testing experience should be a caution to you if you are being told that an investment in extensive GUI-based automation today will have a big payoff when you do localization.

29. The goal of automated localization testing is to show that previously working baseline functionality still works. *(Consensus)*
30. If organized planning for internationalization was done, and if the test team has manually checked the translation of all strings (we don't think this can be automated), and if the test team has manually tested the specific functionality changes made for this localization (again, we don't think this can be efficiently automated), then only a small set of automated tests is required / desirable to check the validity of a localization. The automation provides only a sanity check level test. Beyond that, we are relying on actual use/manual testing by local users. *(7 yes, 1 no.)*
31. If baseline and enabling testing (see #28) is strong enough, the marginal return on making test scripts portable across languages is rarely worthwhile except for a small set of carefully selected scripts. *(5 yes, 0 no.)*
32. If translation / localization was done after the fact, without early design for translatability, then we will need a thorough retesting of everything. *(Consensus)* In this case, the baseline language automation scripts may be of significant value.
33. *We disagreed with the following statement:* It is important to regress all bugs in a localized version and, to the extent done in the baseline version, to extend automated tests to establish the same baseline for each language. *(7 no, 1 yes.)*
34. The kinds of bugs likely to arise during a well-planned localization are unlikely to be detected by baseline regression tests. *(9 yes, 0 no)*.
35. We didn't vote on these points, but I'll insert them here because they provoked thought and discussion (and because I think Marick's on the right track.) Brian Marick suggested that in planning for automated localization testing, we should be thinking in terms of several distinguishable classes of tests. Here are some examples:

- *Language-independent automated tests*, such as (in many but not all cases) printer configuration tests, other configuration / compatibility tests, and tests of compatibility with varying paper sizes.
- *Specific-language automated tests*, if these are worthwhile. If you expect to keep selling new releases of your product in French, German, and Spanish versions, you might find value in creating some French-specific, German-specific, and Spanish-specific automated tests.
- *Most tests that are language specific* will probably best be handled by manual localization testing.
- *International-specific tests* that are handled by automation. These tests check the translatability and localizability of the software. For example, you might provide dummy translations that use strings

that are too long, too short, etc. You might provide text that will be hyphenated differently in different countries.

- *Cheaply localizable tests*. Marick's expectation is that this class is small. However, some tests aren't concerned with the strings used or the screens displayed. For example, stress tests to find memory leaks depend on repeated executions of some functions, but the text and graphics on display might not matter. Localization of some of these tests, to the minimum degree necessary to make them useful, will be easy.

The bottom line is that even if you have an extensive suite of automated tests for an English language product, this might not speed you very much when testing a translation.

Structure of the Los Altos Workshop on Software Testing

This is a process developed by Cem Kaner and Brian Lawrence for technical information-sharing across different groups. It's not new. We adapted it from some academic models and models from other industries.

Our goal is to facilitate sharing of information in depth. We don't see enough of that happening today. Instead, what we see today is that:

- A few people read books and articles on testing. These vary in quality and there is no opportunity to ask questions of the author or challenge the data or the conclusions. We think that the BS factor in some of the "data" is pretty high and that it should be more carefully challenged.
- Some people come to conferences, where they see many short presentations. There is never time for more than a few minutes of discussion. Even in panel sessions, only a few minutes are spent by discussants on each other's opinions or data.
- Another problem at conferences is the promotion of credentials instead of the promotion of data. Someone says, "Capers Jones said X" and we all nod sagely and agree that it must be true. But who knows whether this person understood Capers correctly? Or where Capers got his data? We're mindful of an article by Bill Hetzel on software metrics in which he complained of the difficulty of tracking down the data behind many claims.
- Many people learn techniques from peers. If you stay in one company, you all work your way into a common rut. Maybe you dig out a bit by hiring a pricey consultant or contractor. Somehow you divine from them what people in the other companies that they've worked at do. If the consultant does a good job, you steal some ideas. If not, you reject the ideas along with the consultant. This is expensive and often ineffective.

In GUI test automation, I felt that ideas that were being presented as state of the art at some conferences had been in quiet use for years in some companies (i.e. they were not new) or they had consistently failed or proven false in practice. The level of snake oil in some test automation presentations was unacceptable. Soon after I started to do on-the-road consulting, I realized that many of my clients had strong solutions to isolated pieces of the test automation puzzle, but no strategy for filling in the rest. The state of the art was much higher than the conferences, but it wasn't getting across to the rest of us.

I hosted the first Los Altos Workshop with the goal of creating a situation that allowed for transfer of information about test automation in depth, without the BS.

A meeting like this has to be well managed. As you'll see, there are no structured presentations. This is two days of probing discussion. Lawrence volunteered to facilitate the meeting. Drew Pritsker served as co-host to help with the logistics. Various attendees worked for a few hours as reporters (public note-takers).

There was, and is, no charge to attend the meetings. The organizers pay for the room rental. Breakfast and office supplies are handled on a more-or-less pot luck basis. Meetings are held all day Saturday and most of Sunday, with a social event (dinner, drinks, gossip) on Friday night.

FORMAT OF THE MEETING

The workshops (we're now planning our fourth) are structured as follows:

(1) The meetings run two days.

(2) The meeting is managed by an experienced facilitator (Brian and/or III) and is recorded by the other facilitator or by recording-experienced volunteers.

(3) We restrict the agenda to 1-3 tightly defined topics for discussion. We choose narrow enough topics that we can expect to make significant progress on at least one of them. For example, the topics in the first Workshop were:

```
OBJECTIVE: Develop a list of 3 or 4 practical architectural strategies
for GUI-level automated testing using such tools as QA Partner & Visual
Test. In particular, we want to understand how to develop a body of test
cases that meet the following criteria:

(a)   If the product's user interface changes, almost no work is
      required to make the tests work properly with the modified
      program.

(b)   If the user interface language changes (e.g. English to French),
      little work is required to make the tests work properly.

(c)   If new features are added to the program under tests, the impact
      on existing tests is minimal.

(d)   A year from now, a new tester who is working on the next release
      of the program will be able to use these tests

      (i)      with knowledge of what testing of the program is actually
               being done (i.e. what's actually being covered by this
               testing?)

      (ii)     with the ability to tell whether the program has passed,
               failed, or punted each particular test case.
```

(4) If there are 3 topics, we might decide that we only have enough time for one or two of them. For example, in the first workshop we got so interested in localization and maintenance that we deferred documentation until the February 1998 meeting.

(5) The flow of discussion is:

(a) War stories to provide context. Up to 5 volunteers describe a situation in which they were personally involved. The rest of us ask the storyteller questions, in order to determine what "really" happened or what details were important.

Generally, stories are success stories ("we tried this and it worked because") rather than dismal failure stories, but instructive failures are welcome.

No one screens the stories in advance.

(b) The stories either involve specific presentations of techniques or we schedule another phase of the discussion in which specific techniques are presented and discussed by the participants.

(c) We have a general discussion, looking for common themes, additional lessons learned, etc.

(d) One of us boils down some apparent points of agreement or lessons into a long list of short statements and then we vote on each one. Discussion is allowed. This is an interesting piece of the process. Sometimes we discover during voting that an opinion that all of us think is shared by everyone else is actually held only by one articulate speaker.

The list of statements is a group deliverable, which will probably be published.

(6) Publications

We agreed that any of us can publish the results as we see them. No one is the official reporter of the meeting. We agreed that any materials that are presented to the meeting or developed at the meeting could be posted to any of our web sites. If one of us writes a paper to present at the meeting, everyone else can put it up on their sites. Similarly for flipchart notes, etc. No one has exclusive control over the workshop material.

We also agreed that any publications from the meeting would list all attendees, as contributors to the ideas published.

I believe strongly that this process is valuable, but we can't expand it by inviting more people to the Los Altos meetings. These meetings must be kept small. With two facilitators working together, we can handle about 21 people. Even that feels unwieldy.

The way to expand the process is to train other groups to develop their own collaborative discussions. Accordingly, Brian Lawrence and I make the following offer. If you are trying to set up a discussion like LAWST, on a strictly non-profit basis, with attendees from multiple companies, then Brian and I will be glad to help you set it up. In particular, if you'll reimburse our expenses (our offer is to charge no fee for this service), and if we can work out the calendar dates, then Brian and/or I will come to your first meeting and run it with you. We think/hope that you'll be able to run subsequent meetings (one or two per year) on your own.

References

Arnold, T. (1996) *Software Testing with Visual Test 4.0*, IDG Books.

Bach, J. (1995, October) "Test Automation Snake Oil," *Windows Tech Journal*.

Buwalda, H.J. (1996) "Automated testing with Action Words, Abandoning Record & Playback," *Proceedings of the Eurostar Conference*.

Gause, D. & Lawrence, B. (*in preparation*) *Managing Requirements*, to be published by Dorset House.

Groder, C. (1997) "Building Industrial-Strength GUI Tests," *Proceedings of the 8th International Conference on Software Testing, Analysis & Review*, San Jose, CA., p. 381-96.

Hayes, L.G. (1995) *The Automated Testing Handbook*, Software Testing Institute.

Kaner, C. (1997) "Improving the Maintainability of Automated Test Suites," *Proceedings of the 1997 Quality Week*, San Francisco, CA.

Kaner, C., Falk, J. & Nguyen, H.Q. (1993) *Testing Computer Software*, International Thomson Computer Press.

Kaner, C. & Lawrence, B. (1997) "Los Altos Workshop on Software Testing," Presented at a Birds of a Feather Session at the *Pacific Northwest Software Quality Conference, Portland, OR.*

Marick, B. (1997) "Classic Testing Mistakes" *Proceedings of the 8th International Conference on Software Testing, Analysis & Review*, San Jose, CA.

Pettichord, B. (1996) "Success with Test Automation," *Proceedings of the 1996 Quality Week, Conference*, San Francisco, CA.