Improving the Maintainability of Automated Test Suites[1]

Paper Presented at Quality Week '97

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.

These myths are spread by tool vendors, by executives who don't understand testing, and even by testers and test managers who should (and sometimes do) know better.

Some companies have enjoyed success with these tools, but several companies have failed to use these tools effectively.

In February, thirteen experienced software testers met at the Los Altos Workshop on Software Testing (LAWST)[2] for two days to discuss patterns of success and failure in development of maintainable black box regression test suites. Our focus was pragmatic and experience-based. We started with the recognition that many labs have developed partial solutions to automation problems. Our goal was to pool practical experience, in order to make useful progress in a relatively short time. To keep our productivity high, we worked with a seasoned facilitator (Brian Lawrence), who managed the meeting.

These were the participants: 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.

This paper integrates some highlights of that meeting with some of my other testing experiences.

What's the Problem?

There are many pitfalls in automated regression testing. I list a few here. James Bach (one of the LAWST participants) lists plenty of others, in his paper "Test Automation Snake Oil."[3]

---

[1] Parts of this paper were published in Kaner, C., "Pitfalls and Strategies in Automated Testing" *IEEE Computer*, April, 1997, p. 114-116. I've received comments from several readers of that paper and of a previously circulated draft of this longer one. I particularly thank David Gelperin, Mike Powers, and Chris Adams for specific, useful comments.

[2] A Los Altos Workshop on Software Testing (LAWST) is a two-day meeting that focuses on a difficult testing problem. This paper describes the first LAWST, which was held on February 1-2, 1997. These meetings are kept small and are highly structured in order to encourage participation by each attendee. As the organizer and co-host of the first LAWST, I'll be glad to share my thoughts on the structure and process of meetings like these, with the hope that you'll set up workshops of your own. They are productive meetings. Contact me at `kaner@kaner.com` or check my web page, `www.kaner.com`.

[3] *Windows Tech Journal*, October 1995.

Problems with the basic paradigm:

Here is the basic paradigm for GUI-based automated regression testing:[4]

(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.

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

***First problem*: *this is not cheap*.** It usually takes between 3 and 10 times as long (and can take much longer) to create, verify, and minimally document[5] the automated test as it takes to create and run the test once by hand. Many tests will be worth automating, but for all the tests that you run only once or twice, this approach is inefficient.

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. 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?

***Second problem*: *this approach creates risks of additional costs*.** We all know that the cost of finding and fixing bugs increases over time. As a product gets closer to its (scheduled) ship date more people work with it, as in-house beta users or to create manuals and marketing materials. The later you find and fix significant bugs, the more of these people's time will be wasted. If you spend most of your early testing time writing test scripts, you will delay finding bugs until later, when they are more expensive.

Third problem: these tests are not powerful. The only tests you automate are tests that the program has already passed. How many new bugs will you find this way? The estimates that I've heard range from 6% to 30%. The numbers go up if you count the bugs that you find while creating the test cases, but this is usually manual testing, not related to the ultimate automated tests.

Fourth problem: in practice, many test groups automate only the easy-to-run tests. 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.

Now consider maintainability:

Maintenance requirements don't go away just because your friendly automated tool vendor forgot to mention them. Two routinely recurring issues focused our discussion at the February LAWST meeting.

- When the program's user interface changes, how much work do you have to do to update the test scripts so that they accurately reflect and test the program?

---

[4] A reader suggested that this is a flawed paradigm ("a straw paradigm"). It is flawed, but that's the problem that we're trying to deal with. If you're not testing in a GUI-based environment (we spent most of our time discussing Windows environments), then this paper might not directly apply to your situation. But this paradigm is widely used in the worlds that we test in.

[5] 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.

- When the user interface language changes (such as English to French), how hard is it to revise the scripts so that they accurately reflect and test the program?

We need strategies that we can count on to deal with these issues.

Here are two strategies that don't work:

Creating test cases using a capture tool: The most common way to create test cases is to use the capture feature of your automated test tool. This is absurd.

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.

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.

Programming test cases on an ad hoc basis: 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 the scripts often repeat exact sequences of commands. This approach is just as fragile as the capture/replay.

Strategies for Success

We didn't meet to bemoan the risks associated with using these tools. Some of us have done enough of that on `comp.software.testing` and in other publications. We met because we realized that several labs had made significant progress in dealing with these problems. But information isn't being shared enough. What seems obvious to one lab is advanced thinking to another. It was time to take stock of what we collectively knew, in an environment that made it easy to challenge and clarify each other's ideas.

Here are some suggestions for developing an automated regression test strategy that works:

1. Reset management expectations about the timing of benefits from automation
2. Recognize that test automation development is software development.
3. Use a data-driven architecture.
4. Use a framework-based architecture.
5. Recognize staffing realities.
6. Consider using other types of automation.

1. Reset management expectations about the timing of benefits from automation.

We all agreed that 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. I think that we were surprised 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 benefits are realized in release N. For example:

- 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.
- 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, across releases of the software, then you should seriously thinking about setting your goals for Version N as:

- providing efficient regression testing for Version N in a few specific areas (such as smoke tests and compatibility tests);
- developing scaffolding that will make for broader and more efficient automated testing in Version N+1.

2. Recognize that test automation development is software development.

You can't develop test suites that will survive and be useful in the next release without clear and realistic planning.

You can't develop extensive test suites (which might have more lines of code than the application being tested) without clear and realistic planning.

You can't develop many test suites that will have a low enough maintenance cost to justify their existence over the life of the project without clear and realistic planning.

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.

As we've learned on so many other software development projects, software developers (in this case, the testers) must:

- understand the requirements;
- adopt an architecture that allows us to efficiently develop, integrate, and maintain our 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.

3. Use a data-driven architecture.[6]

In discussing successful projects, we saw two classes of approaches, data-driven design and framework-based design. These can be followed independently, or they can work well together as an integrated approach.

A data-driven example: 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.

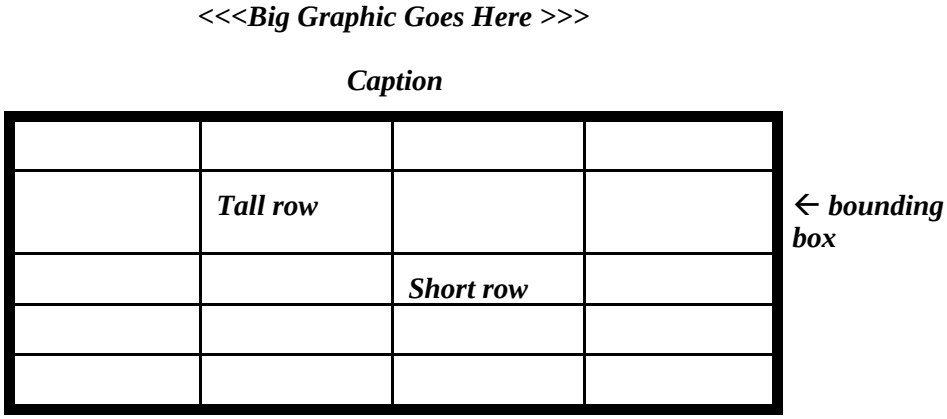

Figure 1 Some characteristics of a table

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.

---

[6] As David Gelperin points out, this paper doesn't reflect on the relationship between the architectural issues that I raise in this paper and the historical development of programming languages. The data-driven architecture does reflect principles underlying 4th generation programming languages. The framework-based architecture does reflect principles underlying 3rd generation programming languages. Many of the ideas in this paper are straightforward applications of good software design. But within the testing community, these ideas are remarkably underused. A broader discussion, that gives readers perspective on the issues, would certainly be a service to the community. My much more limited goal is to raise a few architectural issues in ways that my target readers (working testers and their managers) will find immediately applicable.

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? (Y/N)	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 (one of the LAWST participants) fills his spreadsheet with lists of commands.[7] 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.

4. Use a framework-based architecture.

The *framework* provides an entirely different approach, although it is often used in conjunction with one or more data-driven testing strategies. Tom Arnold (one of the LAWST participants) discusses this approach in his book[8] and courses.

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:

a. 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

---

[7] "Success with Test Automation", *Proceedings of the Quality Week 1996 Conference.*

[8] *Software Testing with Visual Test 4.0,* IDG Books, 1996.

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 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.

b. 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[9] 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.

c. 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.

d. 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.

e. Define utility functions.

---

[9] *Software Testing with Visual Test 4.0,* IDG Books, 1996.

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.

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. Several 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.

5. Recognize staffing realities.

You must educate your management into several staffing issues.

First, many testers are relatively junior programmers. They don't have experience designing systems. Poorly designed frameworks can kill the project too. So can overly ambitious ones. To automate successfully, you may have to add more senior programmers to the testing group.

Second, 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.

Non-programmers can be well served by data-driven approaches that let them develop test cases simply by entering test planning ideas into a spreadsheet.

Third, be cautious about using contractors to implement automation. Develop these skills in-house, using contractors as trainers or to do the more routine work.

Finally, you must educate management that it takes time to automate, and you don't gain back much of that time during the Release 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, 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.

6. Consider using other types of automation

The LAWST meeting focused on GUI-level regression tools and so I have focused on them in this article. Near the start of the LAWST meeting, we each described our experiences with test automation. Several of us had dramatic successes to report, but most of the biggest successes involved extensive collaboration with programmers who were writing the application under test. The types of tools used in these success stories varied widely, reflecting the many different kinds of benefits available from many different kinds of testing tools.

There is too much hype, mythology, and wishful thinking surrounding GUI-based regression automation. They can create an illusion of testing coverage where no significant coverage exists, they can cause serious staff turnover, and they can focus your most skilled staff into designing and maintaining test cases that yield relatively few bugs.

These tools can be genuinely useful, but they require a significant investment, careful planning, trained staff, and great caution.

Appendix

Some of the Conclusions Reached by LAWST Participants[10]

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);

[10] As they are listed in our agreements memos, the statements that we voted on were quite terse. I have categorized them and reworded several of them in order to clarify them for people who weren't at the meeting. I believe, but may occasionally be mistaken, that my rewordings would not change any votes.

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.[11] 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

[11] See Kaner, C., Falk, J, and Nguyen, H.Q., *Testing Computer Software*, 2nd Edition, ITCP, 1993, for more details on function equivalence testing.

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 hod 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:

21. The degree to which you can develop a framework depends on the size / sophistication of your staff. *(Consensus).*
22. 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)*
23. 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.

24. 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)*

25. 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.

26. 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)*

27. 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 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.

27. The goal of automated localization testing is to show that previously working baseline functionality still works. *(Consensus)*

28. 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.)*

29. 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.)*

30. 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.

31. *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.)*

32. 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).*

33. 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.