

# *Black Box Software Testing*

## *(Professional Seminar)*

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

Professor of Computer Sciences  
Florida Institute of Technology

### **Section:28**

## **Managing GUI Automation**

Summer, 2002

Contact Information:

kaner@kaner.com

[www.kaner.com](http://www.kaner.com) (testing website)

[www.badsoftware.com](http://www.badsoftware.com) (legal website)

I grant permission to make digital or hard copies of this work for personal or classroom use, with or without fee, provided that (a) copies are not made or distributed for profit or commercial advantage, (b) copies bear this notice and full citation on the first page, and if you distribute the work in portions, the notice and citation must appear on the first page of each portion, (c) each page bear the notice "Copyright (c) Cem Kaner" or if you changed the page, "Adapted from Notes Provided by Cem Kaner". Abstracting with credit is permitted. The proper citation for this work is Cem Kaner, *A Course in Black Box Software Testing (Professional Version)*, Summer-2002, [www.testing-education.org](http://www.testing-education.org). To copy otherwise, to republish or post on servers, or to distribute to lists requires prior specific permission and a fee. Request permission to republish from [kaner@kaner.com](mailto:kaner@kaner.com).

# *Black Box Software Testing*

---

## **Managing GUI Automation**

### **ACKNOWLEDGEMENT**

Much of the material in this section was developed or polished during the meetings of the Los Altos Workshop on Software Testing (LAWST). See the paper, “Avoiding Shelfware” for lists of the attendees and a description of the LAWST projects.

# *Overview*

---

- **The GUI regression test paradigm**
- **19 common mistakes**
- **27 questions about requirements**
- **Planning for short-term ROI**
- **6 successful architectures**
- **Conclusions from LAWST**
- **Breaking away from the regression paradigm**

# *Overview*

In the mass-market software world, many efforts to automate testing have been expensive failures. Paradigms that dominate the Information Technology and DoD-driven worlds don't apply well to the mass-market paradigm. Our risks are different. Solutions that are highly efficient for those environments are not at all necessarily good for mass-market products.

The point of an automated testing strategy is to save time and money. Or to find more bugs or harder-to-find bugs. The strategy works if it makes you more effective (you find better and more-reproducible bugs) and more efficient (you find the bugs faster and cheaper).

I wrote a lot of test automation code back in the late 1970's and early 1980's. (That's why WordStar hired me as its Testing Technology Team Leader when I came to California in 1983.) Since 1988, I haven't written a line of code. I've been managing other people, and then consulting, seeing talented folks succeed or fail when confronted with similar problems. I'm not an expert in automated test implementation, but I have strengths in testing project management, and how that applies to test automation projects. That level of analysis--the manager's view of a technology and its risks/benefits as an investment--is the point of this section.

# *The Regression Testing Paradigm*

**This is the most commonly discussed automation approach:**

- create a test case
- run it and inspect the output
- if the program fails, report a bug and try again later
- if the program passes the test, save the resulting outputs
- in future tests, run the program and compare the output to the saved results. Report an exception whenever the current output and the saved output don't match.

# *Is this Really Automation?*

Analyze product	--	human
Design test	--	human
Run test 1 <sup>st</sup> time	--	human
Evaluate results	--	human
Report 1 <sup>st</sup> bug	--	human
Save code	--	human
Save result	--	human
Document test	--	human
Re-run the test	--	<b>MACHINE</b>
Evaluate result	--	machine <i>plus</i> <i>human if there's</i> <i>any mismatch</i>
Maintain result	--	human

**Woo-hoo! We  
really get the  
machine to do a  
*whole lot of our*  
work!**

(Maybe, but not  
this way.)

# *GUI Automation is Expensive*

- Test case creation is expensive. Estimates run from 3-5 times the time to create and manually execute a test case (Bender) to 3-10 times (Kaner) to 10 times (Pettichord) or higher (LAWST).
- You usually have to increase the testing staff in order to generate automated tests. Otherwise, how will you achieve the same breadth of testing?
- Your most technically skilled staff are tied up in automation
- Automation can delay testing, adding even more cost (albeit hidden cost.)
- Excessive reliance leads to the 20 questions problem. (Fully defining a test suite in advance, before you know the program's weaknesses, is like playing 20 questions where you have to ask all the questions before you get your first answer.)

# *GUI Automation Pays off Late*

---

- Regression testing has low power because:
  - » Rerunning old tests that the program has passed is less powerful than running new tests.
- Maintainability is a core issue because our main payback is usually in the next release, not this one.

# *Test Automation is Programming*

**Win NT 4 had 6 million lines of code, and 12 million lines of test code**

**Common (and often vendor-recommended) design and programming practices for automated testing are appalling:**

- **Embedded constants**
- No modularity
- ***No source control***
- No documentation
- No requirements analysis
- No wonder we fail.

# *Requirements Analysis*

**Automation requirements are not just about the software under test and its risks. To understand what we're up to, we have to understand:**

- Software under test and its risks
- The development strategy and timeframe for the software under test
- How people will use the software
- What environments the software runs under and their associated risks
- What tools are available in this environment and their capabilities
- The regulatory / required recordkeeping environment
- The attitudes and interests of test group management.
- The overall organizational situation

# *Requirements Analysis*

**Requirement: “Anything that drives design choices.”**

**The paper (Avoiding Shelfware) lists 27 questions. For example,**

*Will the user interface of the application be stable or not?*

- Let’s analyze this. The reality is that, in many companies, the UI changes late.
- Suppose we’re in an extreme case. Does that mean we cannot automate cost effectively? No. It means that we should do only those types of automation that will yield a fast return on investment.

# *You Can Plan for Short Term ROI*

---

- **Smoke testing**
- **Configuration testing**
- **Variations on a theme**
- **Stress testing**
- **Load testing**
- **Life testing**
- **Performance benchmarking**
- **Other tests that extend your reach**

# *Six Sometimes-Successful Test Architectures*

- Quick & dirty
- Equivalence testing
- Framework
- Data-driven
- Application-independent data-driven
- Real-time simulator with event logs

# *Equivalence Testing*

---

**A/B comparison**

**Random tests using an oracle**

**Regression testing is the weakest form**

# *Framework-Based Architecture*

**Frameworks are code libraries that separate routine calls from designed tests.**

- modularity
- reuse of components
- hide design evolution of UI or tool commands
- partial salvation from the custom control problem
- independence of application (the test case) from user interface details (execute using keyboard? Mouse? API?)
- important utilities, such as error recovery

**For more on frameworks, see Linda Hayes' book on automated testing, Tom Arnold's book on Visual Test, and Mark Fewster & Dorothy Graham's book "Software Test Automation."**

# *Data-Driven Tests*

---

- Variables are data
- Commands are data
- UI is data
- Program's state is data
- Test tool's syntax is data

# *Data-Driven Architecture*

- **In test automation, there are three interesting programs:**
  - The software under test (SUT)
  - The automation tool that executes the automated test code
  - The test code (test scripts) that define the individual tests
- **From the point of view of the automation software,**
  - The SUT's variables are data
  - The SUT's commands are data
  - The SUT's UI is data
  - The SUT's state is data
- **Therefore it is entirely fair game to treat these implementation details of the SUT as values assigned to variables of the automation software.**
- **Additionally, we can think of the externally determined (e.g. determined by you) test inputs and expected test results as data.**
- **Additionally, if the automation tool's syntax is subject to change, we might rationally treat the command set as variable data as well.**

# *Data-Driven Architecture*

**In general, we can benefit from separating the treatment of one type of data from another with an eye to:**

- optimizing the maintainability of each
- optimizing the understandability (to the test case creator or maintainer) of the link between the data and whatever inspired those choices of values of the data
- minimizing churn that comes from changes in the UI, the underlying features, the test tool, or the overlying requirements

# *Data-Driven Architecture: Calendar Example*

Imagine testing a calendar-making program.

The look of the calendar, the dates, etc., can all be thought of as being tied to physical examples in the world, rather than being tied to the program. If your collection of cool calendars wouldn't change with changes in the UI of the software under test, then the test data that define the calendar are of a different class from the test data that define the program's features.

- 1. Define the calendars in a table. This table should not be invalidated across calendar program versions. Columns name features settings, each test case is on its own row.**
- 2. An interpreter associates the values in each column with a set of commands (a test script) that execute the value of the cell in a given column/row.**
- 3. The interpreter itself might use “wrapped” functions, i.e. make indirect calls to the automation tool's built-in features.**

# *Data-Driven Architecture: Calendar Example*

This is a good design from the point of view of optimizing for maintainability because it separates out four types of things that can vary independently:

- 1. The descriptions of the calendars themselves come from real-world and can stay stable across program versions.*
- 2. The mapping of calendar element to UI feature will change frequently because the UI will change frequently. The mappings (one per UI element) are written as short, separate functions that can be maintained easily.*
- 3. The short scripts that map calendar elements to the program functions probably call sub-scripts (think of them as library functions) that wrap common program functions. Therefore a fundamental change in the program might lead to a modest change in the software under test.*
- 4. The short scripts that map calendar elements to the program functions probably also call sub-scripts (think of them as library functions) that wrap functions of the automation tool. If the tool syntax changes, maintenance involves changing the wrappers' definitions rather than the scripts.*

# *Data Driven Architecture*

## **Note with this example:**

- we didn't run tests twice
- we automated execution, not evaluation
- we saved SOME time
- we focused the tester on design and results, not execution.

## **Other table-driven cases:**

- automated comparison can be done via a pointer in the table to the file
- the underlying approach runs an interpreter against table entries.
  - » Hans Buwalda and others use this to create a structure that is natural for non-tester subject matter experts to manipulate.

# *Application-Independent Data-Driven*

---

- Generic tables of repetitive types
- Rows for instances
- Automation of exercises

# *Real-time Simulator*

---

**Test embodies rules for activities**

**Stochastic process**

**Possible monitors**

- Code assertions
- Event logs
- State transition maps
- Oracles

## *Think About:*

- Automation is software development.
- Regression automation is expensive and can be inefficient.
- Automation need not be regression--you can run new tests instead of old ones.
- Maintainability is essential.
- Design to your requirements.
- Set management expectations with care.

# *GUI Regression Strategies: Some Papers of Interest*

Chris Agruss, Automating Software Installation Testing

James Bach, Test Automation Snake Oil

Hans Buwalda, Testing Using Action Words

Hans Buwalda, Automated testing with Action Words: Abandoning Record & Playback

Elisabeth Hendrickson, The Difference between Test Automation Failure and Success

Cem Kaner, Avoiding Shelfware: A Manager's View of Automated GUI Testing

John Kent, Advanced Automated Testing Architectures

Bret Pettichord, Success with Test Automation

Bret Pettichord, Seven Steps to Test Automation Success

Keith Zambelich, Totally Data-Driven Automated Testing

