

Black Box Software Testing

2004 Academic Edition

PART 15 -- GUI REGRESSION AUTOMATION

by

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

Professor of Software Engineering

Florida Institute of Technology

and

James Bach

Principal, Satisfice Inc.

These notes are partially based on research that was supported by NSF Grant EIA-0113539 ITR/SY+PE: "Improving the Education of Software Testers." Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Kaner & Bach grant permission to make digital or hard copies of this work for personal or classroom use, including use in commercial courses, provided that (a) Copies are not made or distributed outside of classroom use for profit or commercial advantage, (b) Copies bear this notice and full citation on the front page, and if you distribute the work in portions, the notice and citation must appear on the first page of each portion. Abstracting with credit is permitted. The proper citation for this work is "Black Box Software Testing (Course Notes, Academic Version, 2004) www.testingeducation.org", (c) Each page that you use from this work must bear the notice "Copyright (c) Cem Kaner and James Bach, kaner@kaner.com", or if you modify the page, "Modified slide, originally from Cem Kaner and James Bach", and (d) If a substantial portion of a course that you teach is derived from these notes, advertisements of that course should include the statement, "Partially based on materials provided by Cem Kaner and James Bach." 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 Cem Kaner, kaner@kaner.com.

GUI Regression Automation: Readings

- 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
- Doug Hoffman, Test Automation course notes
- Cem Kaner, Avoiding Shelfware: A Manager's View of Automated GUI Testing
- Cem Kaner, Architectures of Test Automation
- 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

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
- Some successful architectures
- Conclusions from LAWST
- Breaking away from the regression paradigm

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 1st time -- human
- Evaluate results -- human
- Report 1st bug -- human
- Save code -- human
- Save result -- human
- Document test -- human
- Re-run the test -- MACHINE
- Evaluate result -- machine plus human if there's any mismatch
- 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.

Common mistakes in GUI automation

- Don't underestimate the cost of automation.
- Don't underestimate the need for staff training.
- Don't expect to be more productive over the short term.
- Don't spend so much time and effort on regression testing.
- Don't use instability of the code as an excuse.
- Don't put off finding bugs in order to write test cases.
- Don't write simplistic test cases.
- Don't shoot for "100% automation."
- Don't use capture/replay to create tests.
- Don't write isolated scripts in your spare time.

Common mistakes in GUI automation

- Don't create test scripts that won't be easy to maintain over the long term.
- Don't make the code machine-specific.
- Don't fail to treat this as a genuine programming project.
- Don't "forget" to document your work.
- Don't deal unthinkingly with ancestral code.
- Don't give the high-skill work to outsiders.
- Don't insist that all of your testers be programmers.
- Don't put up with bugs and crappy support for the test tool.
- Don't forget to clear up the fantasies that have been spoonfed to your management.

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.

Requirements questions

- Will the user interface of the application be stable or not?
- To what extent are oracles available?
- To what extent are you looking for delayed-fuse bugs (memory leaks, wild pointers, etc.)?
- 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?
- 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?
- Do you expect this product to sell through multiple versions?
- 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?

Requirements questions

- 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?
- 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?
- 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?)
- 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?
- How frequently do you receive new builds of the software?
- Are new builds well tested (integration tests) by the developers before they get to the tester?

Requirements questions

- To what extent have the programming staff used custom controls?
- How likely is it that the next version of your testing tool will have changes in its command syntax and command set?
- What are the logging/reporting capabilities of your tool? Do you have to build these in?
- 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-synchronize the product and the test (get them operating at the same state in the same program).
- (In general, what kind of functionality will you have to add to the tool to make it usable?)
- Is the quality of your product driven primarily by regulatory or liability considerations or by market forces (competition)?
- Is your company subject to a legal requirement that test cases be demonstrable?

Requirements questions

- Will you have to be able to trace test cases back to customer requirements and to show that each requirement has associated test cases?
- Is your company subject to audits or inspections by organizations that prefer to see extensive regression testing?
- 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?
- What are the skills of your current staff?
- Do you have to make it possible for non-programmers to create automated test cases?
- 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?
- What kinds of tests are really *hard* in your application? How would automation make these tests easier to conduct?

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

Four sometimes-successful GUI regression architectures

- Quick & dirty
- Equivalence testing
- Framework
- Data-driven

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.
 - Ward Cunningham's FIT tool enables the same type of testing, but below the level of the UI. See <http://fit.c2.com/>

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.