The Practical Organization of Automated Software Testing

by Herbert M. Isenberg Ph.D.

Abstract

The purpose of this paper is to take a practical approach to automated software testing and explain requirements for its success. To be successful one needs remember that there are four interrelated components that have to work together and support one another: 1) An automated software testing system based on one point maintenance and reusable modules, 2) Testing infrastructure consisting of the events, tasks and processes that immediately support automated, as well as manual, software testing, 3) Software testing life cycle that defines a set of phases outlining what testing activities to do and when to do them, and 4) Corporate support for repeatable processes. These components are discussed from the point of view of the author's many years experience as a senior software test automation engineer and QA Architect working in a variety of software development environments.

Introduction

The purpose of this paper is to explain how to succeed in automated software testing. To be successful one must remember that there are several interrelated components that have to work together and support one another.

This paper will lay out what these necessary components are and their interrelationships. The emphasis here is on what is practical, what is useful, and what works in the author's experience. Automation is not an island unto itself. It requires a solid testing infrastructure and a thoughtful software testing life cycle that are both supported and valued by the corporate culture.

To begin, there is the automated testing system itself. It must be designed to support reusable module and one point maintenance. It must be very flexible and easy to update. The testing infrastructure includes a dedicated test lab, a good bug tracking system, standard test case format, and comprehensive test plans. The software testing life cycle defines when tasks associated with automation occur in relation to the overall software testing effort that is mostly manual.

A company can make great strides using test automation. The important benefits include, higher test coverage levels, greater reliability, shorted test cycles, ability to do multi user testing at no extra cost, all resulting in increased levels of confidence in the software.

The good news is that when automated testing is introduced each of these components does not need to be in place. Pieces can be introduced gradually into the culture. What is important is the commitment to automation and an understanding of where automation can take a company.

Automation success is a very practical business. The remainder of this paper is dedicated to resolving two recurrent issues confronted by those involved in automated software testing:

- **Issue 1.** How does one design and implement an effective automated software testing system in a Rapid Application Development environment where the interfaces (windows/controls) are continually changing and the data/content is continually being revised and modified ?
- **Issue 2.** Why does automated software testing fail in most corporations?

Issue 1

The issue of "How to build a useful automated software testing system in a constantly changing RAD environment…" is resolved by designing the automated testing system with a set of Architecture Principles based on a Structured Methodology for building 1) Reusable modules and 2) One point system maintenance.

One of the most useful outcomes of this approach is the ability of the automated testing system to change as easily and quickly as the software under test. From a corporate point of view, this is one of the key measures of success for automated testing (and rightly so). For purposes of brevity, the development of an automated software testing design under the methodology of the architectural principles noted above will be referred to as a Practical Automated Testing System or PATS.

How does a PATS work ?
This section focuses on the automated testing system itself, looks at how to build one that is simple, easy to use, supports reusable module and one point maintenance. In other words, a PATS.

Tool Selection
First conduct a tool evaluation. Be sure to have the people who are going to use the tool do the evaluation. I suggest picking two or three tools, set them up side by side, and build a simple and complex test case in each one. Base the evaluation on ease of use and the tools ability to create reusable modules. Remember, each tool has a built in point of view on how to test. The most important point here is to not let the tool tell you how to test. The automated testing methodology is independent of the tool and the tools main role is to support that methodology.

A horse will lead you to water, if you let it. That's fine if you want to trot around the lake, however, its not so good if you want to go across the meadow. The same with automated testing tools. Out of the box, using Capture-playback, you will be taken on a ride to some wild woodland—not where you want to go. Your automated testing system will end up a big entwined mess that is impossible to maintain. This is why it is of the utmost importance that a structured testing methodology is brought to bear on the tool. The testing tool is our servant, not our master.

Automated Testing Methodology: Reusable modules
The basic building block is a reusable module. These modules are used for navigation, manipulating controls, data entry, data validation, error identification (hard or soft), and writing out logs. Reusable modules consist of commands, logic, and data. They should be grouped together in ways that make sense. Generic modules used throughout the testing system, such as initialization and setup functionality, are generally grouped together in files named to reflect their primarily function, such as "Init" or "setup". Others that are more application specific, designed to service controls on a customer window, for example, are also grouped together and named similarly.

The system structure evolves by the way reusable modules are organized. Experience has shown that the most structured and practical way to organize the majority of reusable modules is around an application's windows, or screens. All the modules that service the customer screen are organized in one file, or library. That way, when the customer screen is modified for any reason, updates to the testing system are located in one place, hence the principle of One Point Maintenance comes into existence.

It may be argued that if a control, such as a list box, on the customer screen is similar to one on the inventory screen, why not use one reusable module for both ? This may be trickier to pull off than it

seems, and the level of complexity may not warrant it. As a module becomes more complex, it becomes more difficult to maintain, and there is a greater likelihood of bugs showing up (see McCabe on Cyclomatic complexity).

Test Cases
The next step in the methodology is to turn reusable modules into automated test cases. Here, the automation engineer takes a well-structured manual test case and begins scripting it out, creating reusable modules as he goes. The goal here is to build the reusable modules in a very methodical manner. The action-response pairs from the test case are scripted into reusable modules. These pairs also determine the size or granularity of a reusable module, which generally consist of just one action-response pair. This becomes important later, for determining the scope and cost of the automation project. A single reusable module generally takes one to three hours to script and carefully unit test. The average automated test case requires around twenty to thirty reusable modules to complete.

Automating test cases in this manner allows the testing system to take on a very predictable structure. One benefit of this predictability is the ability to begin building the automated testing system from the requirements early in the software testing life cycle.

Structured Test Case Format
A structured test case format clearly represents the dynamic nature of test cases, consisting of a sequence of action-response pairs.

Test case example;
Test Case ID: CUST.01**;**
Function: Add a new Customer
Data Assumptions: Customer database has been restored
General Description: Add a new customer, via the Customer Add screen, and validate that the new Customer was displayed corrected on the All Customer Screen.

Action	**Initial State or Screen**	**Data**	**Expected Results (Response)**
1. Bring up Sales application via the Windows Icon	Program Manager	None	Main Sales Application Menu displayed
2. From Views Menu Select Customer.	Main Sales Screen	None	Customer Views Screen Displayed.
3. Click on All Customers	Customer View	None	Customer window is displayed with title “All Customers”.
4. Click on Add Button	Customer - All Customer	None	Add Customer Screen is displayed.
5. Enter data to add a new customer and single click the add button.	Add Customer	Name: John Doe Addr: 123 Main st. City: San Francisco (Or refer to data sheet: Data Sheet - #105)	Data display in indicated fields. (Or: as defined by data sheet.)

Benefits:
1. Easier to automated - all have same structure
2. Data requirements are clearly defined
3. Navigation is precise
4. Expected results for each action is specific - no guess work involved

Predictable Structure
Next, we will examine an example of what is meant by a predictable structure. Each screen named in the test case points to a file containing a set of reusable modules. For example, if the application has thirty different screens, then the testing system will have thirty files named after those screens containing reusable modules specific to each screen. Upon executing a command intended to display a specific screen, the test case's first action is to validate that the intended screen is up, residing on the desk top, (and additionally that the screen is enabled and/or in focus - depending on the specifics of what's being tested). This is accomplished by indexing into the file and calling the appropriate reusable module. In this case, it will be the first module in the file, which is part of the standard for validating a new screen. The reusable module knows how to 1) Dynamically validate the title of expected screen using multi level validation methods, 2) Assign an error level if the validation fails and 3) Write a message to a detailed log file. Note: I suggest at least two log files, a detail log and a one line pass/fail log.

Here is an example of indexing into the customer file and calling the reusable object to validate that the customer screen is displayed. In this example, the index is the label "VALIDATE":

Comments	Actions
Validate the customer screen is up	CALL "V_CUSTOMER" label "VALIDATE"

Here is an example of a reusable module that 1) Dynamically validates the customer screen using multi level validation 2) contains logic to assign error levels and 3) writes out two messages to the detail log.

Comments	Actions
File index to validate customer screen	Label "VALIDATE:"
Write test case context along with date/time to detailed log for debugging purposes	Write $CURLOG LOG.CUR_OTL
Validate the text on the customer screen using multi level validation.	Look text CONSTANTS.CUST_TXT win CUST.TITLE_WIN area wait 1
Validation logic Log error level as a soft error Increment soft error counter - keep going resume to calling script Logs pass/fail messages.	If no >Log $CURLOG "Soft Error: Did not find customer screen" >Assign ERROR.SOFT=ERROR.SOFT+"1" >Resume Otherwise >log $CURLOG "Passed: Customer screen is displayed" >resume

(code example written in 4GL language native to AutoTester toolset by Software Recording Inc.)

Storing the text string value in the variable CONSTANTS.CUST_TXT, supports the principle of One Point Maintenance. This is a practical way of storing text validation strings so they are easy to locate and update.

It is possible to increase or deepen the principle of One Point Maintenance, allowing for more seamless cross platform testing, or portability, through the use of generic screen and application variables. This technique allows the automated testing system to handle dynamically changing windows which is often required in a RAD environment. This increases the systems overall flexibility and reduces maintenance time.

This structured method of building reusable modules continues in this same manner for each test case, resulting in a predictable, well-structured and most practical, automated testing system. Like a person laying bricks to build a house, one methodically builds reusable modules to express test case functionality. In the final analysis, this is what constitutes a Practical Automated Testing System (PATS).

Multi Level Validation
Multi level data validation increases the usefulness and flexibility of the testing system. The more levels of data validation, or evaluation, the more flexible the testing system. Multi level validation and evaluation is the ability of the testing system to 1) Perform dynamic data validation at multiple levels and 2) Collect information from system messages so that others can evaluate the data at some later point in time.

Dynamic data validation is the process whereby the automated testing tool, in real time, gets data from a control, compares it to an expected value and writes the result to a log file. It also expresses the ability of the testing tool to make branching decisions based on the outcome of the comparison. Ideally, the testing tool has the ability to combine the GET and COMPARE into one command, such as a LOOK function, to simplify coding.

Validation refers to correctness of data decided dynamically. Evaluation refers to system messages that will be collected but evaluated after the fact.

In a PATS, dynamic data validations and message evaluations can be conducted at seven different levels with two compare modes, inclusive and exclusive.

Having previously pointed out that an automated testing system is not an island unto itself, let’s continue with the section which addresses the reason why automated software testing most often fails.

Issue 2

To avoid the pitfalls commonly experienced by corporations implementing automated testing these elements must be carefully examined: 1) The testing infrastructure, 2) The software testing life cycle and 3) Corporate support.

The Testing Infrastructure
The testing infrastructure consists of the testing activities, events, tasks and processes that immediately support automated, as well as manual, software testing. The stronger the infrastructure the more it provides for stability, continuity and reliability of the automated testing process.
The testing infrastructure includes:

- Test Plan
- Test cases

- Baseline test data
- A process to refresh or roll back baseline data
- Dedicated test environment, i.e. stable back end and front end
- Dedicated test lab
- Integration group and process
- Test case database, to track and update both automated and manual tests
- A way to prioritize, or rank, test cases per test cycle
- Coverage analysis metrics and process
- Defect tracking database
- Risk management metrics/process (McCabe tools if possible)
- Version control system
- Configuration management process
- A method/process/tool for tracing requirement to test cases
- Metrics to measure improvement

The testing infrastructure serves many purposes, such as:

- Having a place to run automated tests, in unattended mode, on a regular basis
- A dedicated test environment to prevent conflicts between on-going manual and automated testing
- A process for tracking results of test cases, both those that pass or fail
- A way of reporting test coverage levels
- Ensuring that expected results remain consistent across test runs
- A test lab with dedicated machines for automation enables a single automation test suite to conduct multi-user and stress testing

It is important to remember that it is not necessary to have all the infrastructure components in place in the beginning to be successful. Prioritize the list, add components gradually, over time, so the current culture has time to adapt and integrate the changes. Experience has proven it takes an entire year to integrate one major process, plus one or two minor components into the culture.
Again, based on experience, start by creating a dedicated test environment and standardizing test plans and test cases. This, along with a well-structured automated testing system will go a long way toward succeeding in automation.

The Software Testing Life Cycle

The Software Testing Life Cycle, (STLC), is the road map to automation success. It consists of a set of phases that define what testing activities to do and when to do them. It also enables communication and synchronization between the various groups that have input to the overall testing process. In the best of worlds the STLC parallels the Software Development Life Cycle, coordinating activities, thus providing the vehicle for a close working relationship between testing and development departments.

The following is a “meat-and-potatoes” list of names for the phases of the STLC:

1. Planning
2. Analysis
3. Design
4. Construction
5. Testing - Initial test cycles, bug fixes and re-testing
6. Final Testing and Implementation
7. Post Implementation

Each phase defines five to twenty high level testing tasks or processes to prepare and execute both manual and automated testing. A few examples are in order:

1. Planning
    - Marketing group writes a document defining the product
    - Define problem reporting procedures
    - High level test plan
    - Begin analyzing scope of project
    - Identify acceptance criteria
    - Setup automated test environment
2. Analysis
    - Marketing and Development groups work together to write a product requirements document
    - Develop functional validation matrices based on business requirements
    - Identify which test cases make sense to automate
    - Setup database to track components of the automated testing system, i.e. reusable modules
    - Map baseline data to test cases
3. Design
    - Development group writes a detailed document defining the architecture of the product
    - Revise test plan and cases based on changes
    - Revisit test cycle matrices and timelines
    - Develop risk assessment criteria (McCabe tools help here)
    - Formalize details of the automated testing system, i.e. file naming conventions and variables
    - Decide if any set of test cases to be automated lend themselves to a data driven/template model
    - Begin scripting automated test cases and building reusable modules

As the STLC is continually refined it will spell out the organization of the testing process: What steps need to be taken and when, to ensure that when the software is ready to test, both the manual and automated testing system will be in place and ready to go. The idea here is to start early and be ready to respond to change.

One of the biggest reasons automation fails is the lack of preparation early in the process. This is due in part to a lack of understanding of what needs to be done and when. The steps are not difficult, it is just a matter of understanding how the STLC works. It does not take any more time and effort to succeed than it does to fail.

As with the Testing Infrastructure, it is not necessary to have all the STLC tasks in place to succeed with both manual and automated testing. The most important lesson many of us have learned is “Start Early!” A great deal of automation work can be done before the software is available, if one is working with a well-structured, one point maintenance, automated testing system and methodology, i.e. PATS.

The Corporation

Corporation dynamics are an entire field unto themselves. The critical point regarding the corporation, for the purposes of automated software testing, is that there must be a commitment to adopting and supporting repeatable processes. Automation cannot succeed without this commitment.

Conclusion
Practical automated software testing systems is an evolving technology aimed at increasing longevity and reducing maintenance.

Longevity reflects the automated testing systems ability to intelligently adjust and respond to unexpected changes to the application under test. By competently doing so, the testing system's life expectancy naturally increases through its inherent ability to stay useful over time.

Reduced maintenance saves time. This is a function of the elimination of redundancy, exemplified by features such as "One Point" maintenance and reusable modules.

PATS is designed to provide solutions to testing challenges arising from new development technologies and environments. It is important to remember that testing systems execute test cases, they do not define or prioritize them. It takes a robust testing infrastructure and an intelligent STLC to support the practice of automated testing and establish the criteria for success.

I would like to end with a couple of reminder lists:

Practical features of automated software testing systems:

- Run all day and night in unattended mode
- System continues running even if a test case fails
- Keep the automated system up and running at all costs
- Recognize the difference between hard and soft errors
- Write out meaningful logs
- One point maintenance
- Easy to update reusable modules
- Text strings stored in variables easy to find and update
- Written in an English-like language easy to understand
- Automated most important business functions first.
- Quickly add scripts and modules to the system for new features
- Don't waste time with very complex features, keep it simple
- Collect other useful information such as operating system and CASE tool message
- Track components of the automated testing system in a database
- Track reusable modules to prevent redundancy
- Carefully test the testing system !
- Keep track of tests coverage provide by automated test suites
- Track which test cases are automated and which are manual
- Use same architecture for Web or GUI based application testing
- Make sure baseline data is defined and process in place to refresh data
- Keep test environment clean and up-to-date
- Test case management - store test cases in a database for maintenance purposes
- Track tests that pass, as well as test that fail.

Automation - How to keep it going

- Test Lab with dedicated machines
- Run test daily
- Have it become an active part of the culture
- Demonstrate the usefulness of automation to other group such as development and implementation.

- Get developers interested in the automated testing by running test for them.
- Build a suite of test specifically for developers or end users

Key words: Automation, testing, QA, infrastructure, life cycle, reusable, automated testing, RAD.

Author Information

Herbert M. Isenberg Ph.D.
Technical Director of Automated Testing/Quality Assurance Architect
Oacis Healthcare Systems
PO Box 3178
Sausalito, CA. 94966

Type: Experience Report

Phone:
(ofc) 415.667.0878
(hm) 415.331.1071
Fax: 415.667.9644
Email: hisen@slip.net

Biographical Information

Herb Isenberg Ph.D, is the sponsor of the Web site http://www.automated-testing.com . He has more than 17 years experience in the field of Quality Assurance and Automated Testing. He is often quoted in articles on automated testing, has published in Dr. Dobbs Journal as well as being a frequent conference speaker on all aspects of automated testing. He currently is a Technical Director and the QA Manager for the Mutual Funds Group at Charles Schwab in San Francisco. Other companies he has worked for include Sybase, Ingress, Matson Navigation, and Oacis Healthcare Systems as their QA Architect. His Ph.D. is in Medical Sociology from University of California San Diego and post-doctoral work is in Doctor-Patient interactions and Natural Language sequencing in the field Conversational Analysis.