



sqaMethods Approach to Building Testing Automation Systems

By
Leopoldo A. Gonzalez
leopoldo@sqaMethods.com

<u>BUILDING A TESTING AUTOMATION SYSTEM</u>	3
<u>OVERVIEW</u>	3
<u>GOALS FOR AN AUTOMATION SYSTEM</u>	3
<u>BEGIN WITH THE END IN MIND</u>	4
<u>THE THREE TIER SYSTEM</u>	5
<u>APPLYING THE CONCEPT</u>	6
<u>BREAKING DOWN THE COMPLEXITY</u>	8
<u>CONCLUSION</u>	10

Building a Testing Automation System

Overview

While there may be many ways of building a testing automation system and most implementations can perform a certain level of automation, I believe that ultimately the automation system should give testers the capability of an easy transition from a manual test script to an automated one. It is up to the automation engineer to hide the complexities of code behind an elegant set of instructions that manual testers can understand.

This paper is a collection of thoughts and approaches that describe my experience in building such a system.

Goals for an Automation System

Early in my testing automation career I realized that in order for an automation system to be successful, it had to provide five basic functions. Without these capabilities, the automation system would lack in flexibility and reliability.

1. *The automated script must be able to be executed as a single entity.*

This simply means that the tester must be able to pick a script from a list of scripts and choose to run it alone. The script must know that it is not running as part of a suite and if needed, be able to gather its own information.

2. *The automated script must be able to be executed as a suite.*

The same script must know that when running as part of a suite, it cannot stop and gather information from the user as it would defeat the purposed of running unattended. Also there are times when you don't want the script to exit the application under test only to be re-launched by the next script in the

suite. This is a waste of time and only prolongs the execution time. The script must know not to exit but to return to a **HOME** state.

3. *The script must be data driven.*

The data the script uses must reside somewhere other than in the script itself. There should never be hard coded data inside the script. Data driven scripts allow the tester to create multiple test case scenarios without having to modify the code itself.

4. *All of the test scripts in a suite must execute.*

This means that the failure of one script should not affect the execution of the next one. If during the execution of a script, the application aborts, the script must be able to recover from that failure and attempt to continue with the next script in the suite.

5. *The automated script should use the same logic flow as the manual script*

The transition between a manual script and an automated one will be a lot easier if when converting a script, the tester can follow the same logic flow and use some of the same language he used before.

While each of the above goals is worthy of its own analysis, this paper will not cover all five goals, however I will talk a little on the last one.

Begin with the End in mind

Let's begin with a visualization of what a testing automation system should be for a QA shop. This system should be treated as a useful tool to be used during the QA test cycle. It should be the first line of tests to be executed when a new build is created (smoke test). It should be used as a regression test mechanism (regression test), it should be updated with new test cases for new functionality and finally, if performance test is included, it should be used to measure the system's performance under varying loads.

But the question is, how do get from where you are to a system that is part of a well oiled QA testing machine?

The Three Tier System

The three tier system of software development is very well known and utilized for big projects. It is modular; it separates functions at their right domain area and is just about the right size to avoid too much complexity.

By the same token, in my experience I found out that this model works fine for testing automation systems as well. The only difference is that rather than breaking the model down to SQL, Business Objects and GUI as in the traditional three tier approach, a testing automation system should be made out of Common Utilities, Business Objects and Test Cases.

The methods in the Common Utilities would be inherited by the Business Objects, and these would be inherited by the Test Case scripts. As illustrated in Figure 1.

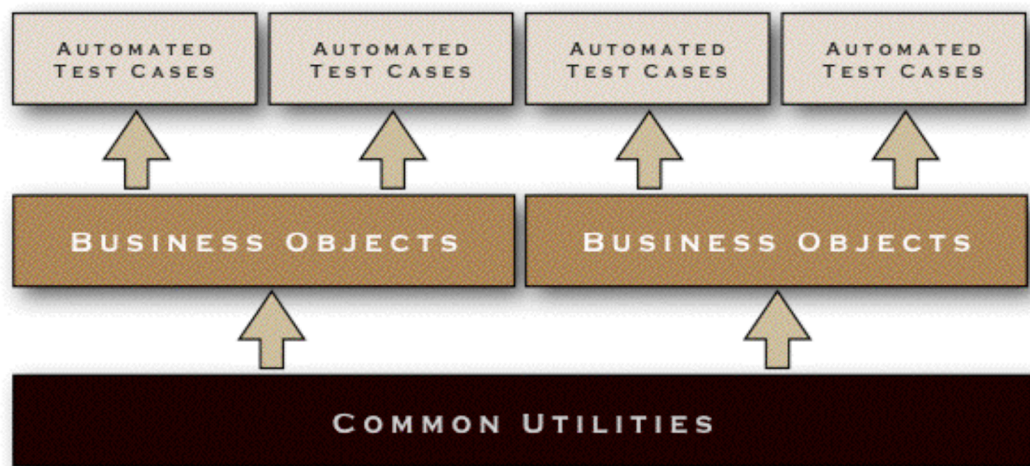


Figure 1

This model allows us to grow capability at the right level of abstraction. If I need a method that would be useful to all of the automation system, I would write it at

the Common Utilities tier. However, if the method falls inside one of the business domains, such as ordering, navigation, admin, etc., then I would write it at the Business Object tier. The last tier to build is the actual test script. The syntax used by the test script should follow closely the logic used in the manual test script. Be aware that you will not be able to completely remove the flavor of the language from your testing automation tool, but at least you will have a system on which a tester can venture out and try to develop an automated script by himself even though he may not be a programmer.

Applying the Concept

To see the benefit of this model, let's examine a manual test case and create the necessary components that will allow us to mirror the syntax of the manual test case. The typical format of a manual test is as follows:

Test Step	Expected Outcome	Actual Outcome	Result
Launch the Internet Explorer	The system should display the internet browser		Pass / Fail
Login to the system by accessing URL "www.mytest.com"	The system should display the login page		Pass / Fail
Verify the layout of the login page	The page should display elements according to the agreed style guide ...		Pass / Fail
Login as customer user "Bob"	The system should display the order page		Pass / Fail
Etc ...			

Table 1

Interestingly enough, IBM Manual Tester ® implements a different format for manual testing that deserves a second look. The instructions above are flattened out and the tool provides the ability to either place it in the log via the Verification Point attribute or skip it by assuming it was a test step. This flat format allows the creation of multiple steps for a single verification point or vice versa; a single step with multiple verification points. In the case of step #3 "Verify the layout of the page", the system

would allow you to enter many verification points for the many objects displayed on the screen. Table 1 above table would read something like Figure 2:









<ul style="list-style-type: none"> Launch Internet Explorer The system should display the internet browser Login to the system by accessing URL "www.mytest.com" The system should display the login page Verify the layout of the page The page should display the elements according to the agreed style guide. Login as customer user "Bob". The system should display the order page.
--

Figure 2

The goal I set for myself when building a testing automation system is to closely match the syntax of the flatten format. I understand that while I may never get to the point where it reads the same, I can certainly copy the flow of the manual test case by making calls to the methods that give me that functionality. Figure 3 demonstrates this idea.

```
<java headers and declarations ...>

public void testMain(Object[] args) {
    TResult tr = new TResult();

    // -----
    // Launch the browser to the URL provided
    tr = launchBrowser("http://www.sqaMethods.com");

    // If the launchBrowser method fails for whatever reason, then we need to log it and exit
    if (tr.getTestVerdict() == FAILED) {
        logMessage(FAILED, "Sorry, I could not launch the browser ");
        System.exit(0);
    }

    // Object tr needs to be cleared out
    tr = null;

    // -----
    // Verify the components in the header
    cellwin().performTest(HeaderElementsVP());

    // Verify the contents of the main page
    tabletable().performTest(MainPageVP());

    // -----
    // Login user type USER Bob
    tr = loginUser("Bob", USER);

    // If the loginUser method failed for whatever reason, then we need to log it and exit
    if (tr.getTestVerdict() == FAILED) {
        logMessage(FAILED, "Sorry, I could not log user Bob", TRUE);
        System.exit(0);
    }
    tr = null;
}
}
```

Figure 3

Don't be taken back by the amount of code necessary to produce the same result as the manual script; after all we are creating a testing automation system, and that requires some coding. What I do want to point out is that the steps from the manual test have been replicated by the sections bounded by the dashes. The code is readable because the methods are named after the words used by the manual tester.

Breaking down the Complexity

In spite of the Java flavor of the script shown in Figure 3, the script hides a lot of infrastructure work that allows the testing automation system to run smoothly. For

example, the `launchBrowser` method hides that if the browser cannot be launched, it will return a verdict of **FAILED**. I can then instruct the script not to bother any more and exit gracefully.

Notice that in the process of launching the browser, I created an object of type `TestResult`, this object contains the typical results gathered during a test or during the execution of a method. A test can result in a `Pass/Fail` state and with either result; it could generate data gathered during the process. Inside the object type `TestResult`, there is a field that will hold the value of the data gathered. In this example I do not show that capability but I bring it up to illustrate that the object is flexible enough to hold other data types if needed.

The next sections are the verification points that check for the properties of the header and the main html document. I don't write custom code in these sections because I believe that the Verification Point feature from the automated tool does a better job in checking these properties than I could. Also I don't do anything with the results of the verification point tests, I let the tool log it and then continue.

Let's look at the `loginUser` method, at first glance it appears that this method is not that complicated, after all, how hard is it to log Bob in? However, when you look at how this method has been implemented you will realize that the concept of inheritance is well at work. The `loginUser` was implemented at the Business Object layer and is part of the `Login` class.

The `Login` class was inherited by the automated script and therefore all of the methods defined in that class are now part of the automated script. In turn, the `loginUser` method utilizes methods from the Common Utilities layer also via inheritance. These are the `readData` and the `loadDriver` methods that take the name of "Bob" and perform a database read to the table "Users"; where Bob and his password are stored.

In this case, I don't consider that hard coding the name of Bob as breaking my own rules since using the name makes the script a lot more readable. Trying to abstract the name would only add more layers of unnecessary complexity.

Notice that throughout the script I check for the state of my `TestObject tr`, and if it fails, then I write it to the log. The `logMessage` method is implemented at the Common Utilities layer and allows the tester to write items to the log. The log can even accept a parameter to take a snapshot of the screen to capture the moment the script failed.

As you can see, trying to replicate the logic flow from the manual test case can be a bit challenging since at the same time, I'm trying to stay true to my own goals. The code sample above provides just a glimpse of how this can be accomplished, and while it did not show the other features, it provides an idea as to how I go about architecting a testing automation system.

Conclusion

Developing a testing automation system should be approached with the same care and analysis as any other software development. However, the automation engineer must not lose sight that the system he/she is building needs to provide a solution to the QA team as quickly as possible. Building a system for the sake of creating something "cool" harms the company and the testing automation industry.

Therefore, in choosing whom you trust to develop your testing automation system should be a top priority. Make sure that the person you hire has the ability to articulate his/her vision and approach to developing an automation system and that he/she has experience doing it.

For more information on how to implement this type of automation system in your company, visit us at www.sqamethods.com or contact me at leopoldo@sqaMethods.com.

About Leopoldo Gonzalez



I started my career in the Software Development industry in 1984. During this time I have performed duties as a Developer, Tester, User Group Test Coordinator and Testing Automation Architect.

I have compiled my experiences, best practices and practical tools in a workshop I call "**Software Testing 101 - What Every Tester Needs To Know About Software QA**". This workshop is aimed at individuals who are new to the world of Software Testing and QA, but also has valuable information for seasoned Testers such as the subject discussed in this paper.

For more information on my workshop, visit us at www.sqaMethods.com