Preventing Skunky Test Automation Shelfware: A Selenium-WebDriver Case Study

Alan Ark

Eid Passport

aark@eidpassport.com

Abstract

Eid Passport had a suite of Selenium tests with a bad reputation—difficult to maintain, broken all the time, and just plain unreliable. A tester would spend more than four days to get through one execution and validation pass of these automated tests. Eid Passport was ready to toss these tests into the trash. Alan Ark volunteered to take a look at the tests with an eye toward showing that Selenium-based tests can, in fact, be reliable and used in the regression test effort. Alan shares techniques he used to transform a sick, test automation codebase into a reliable workhorse. These techniques include AJAX-proofing, use of the Page Object model, and pop-up handling. The test process that used to take more than four days to turnaround now finishes in under two hours. And this is just the beginning.

Biography

A senior SDET with Eid Passport, **Alan Ark** helps the test organization cultivate and develop its automation solutions. Alan draws on more than twenty years of experience in various software quality roles when creating automated solutions to testing problems. Several of his projects have been presented at Quality Week (1999) and PNSQC (2008, 2011, 2013). These projects include validating a buggy data conversion of historical stock trading data into the Euro, championing the use of automated testing in new organizations, and creating testing frameworks from scratch. See Alan's LinkedIn profile at https://www.linkedin.com/in/arkie.

1 Introduction

Automated Graphical User Interface (GUI) testing is currently being used at most companies to help with their regression testing needs. A major challenge with relying on this method for regression testing is the maintenance. As the User Interface (UI) changes, the GUI automation must change with it, or else there is a good chance that the tests will not run correctly.

Improperly implemented GUI automation projects can turn into a maintenance nightmare. Sometimes, the burden of maintaining the automated tests can be so harrowing that they are tossed aside. The result is usually shelfware — software that sits on the shelf, not providing any real value to the engineering organization.

This paper looks at one Selenium-Webdriver based UI automation project that almost turned into shelfware. I took lessons learned from my previous experiences with web based test automation with Web Application Testing in Ruby (WATiR) and applied them here at Eid Passport.

The result is a suite of tests that are reliable and depended upon to give the development team confidence in any build of the software under test.

2 Help!

At Eid Passport, there was a full-time automation engineer who was in charge on the Selenium based test automation. He was a full time employee who was tasked with maintenance and creation of new tests that needed to be implemented. When he left the company, the automated tests were not actively maintained. The upkeep was a task that was done as time permitted by anyone who had extra cycles.

As more people started to run the regression tests, they noticed several patterns that were worrisome. The biggest worry was that the tests would appear to fail randomly. When taking a failed test and running thru it manually, it would always pass. Random tests would fail randomly during nightly runs. The problem was exacerbated by running the tests on different machines. If we are seeing random failures during regression test runs, how can the team have any confidence in the quality of the software?

Another point of concern was the amount of time that it took to get through one pass of the regression tests. Ninety tests took over 4 hours to run. The team was thinking about not using the automated GUI based regression tests anymore due to these two issues.

I volunteered to do a code analysis to see if the above issues could be addressed. I had championed the use of Page Objects in this automation suite, so I knew that any modifications to the code could be made in a confident matter. During the analysis phase, I noticed that the automation suite did not handle the AJAX parts of our application very well – which would result in the non-reproducible errors that were haunting the test suite. I felt that I could use the majority of the existing code, but tweak some of the underlying methods to make them more reliable. I also felt that with some optimizations, the test suite would finish in less time as well.

3 How Did We Get Here?

The Selenium based tests were actually version 2 of the GUI regression test suite for Eid Passport. The original version was implemented in Microsoft Coded UI and did not use a Page Object model. Using Microsoft Coded UI locked in the browser that you are using to Internet Explorer (IE). At the time, using any other browser for running the test automation would be out of the question. Microsoft has since remedied this to allow coded UI tests to be run on multiple browsers, but this was not available at the time of original implementation. By not using a Page Object model, the original test suite was difficult to maintain as UI changes were made to the system under test.

I discussed the pros and cons with the automation engineer who was tasked with maintaining the regression tests and gave him an overview of a new design using Selenium. The biggest con to the project would be that he would have to reimplement the test suite. But moving to a new solution based on Selenium-WebDriver would give us the following advantages. The WebDriver API is now a part of the World Wide Web Consortium (W3C). By using Selenium-WebDriver, the company gained much more flexibility into the browsers that could be used. The support ecosystem for Selenium is also richer as it has gained popularity across the world. This also means that it would be easier to find new employees who could hit the ground running upon hire. By using the Selenium libraries, we had a choice in what language to use. The fact that Eid Passport is a Microsoft shop coupled with Selenium's good support for coders in C# made it a natural choice. By using Page Objects, the test suite would be much easier to maintain.

These pros readily outweighed the cons. He got the green light! Onward with the Selenium based tests.

4 Analysis of the Project

After spending some time looking at the code, I came up with a plan of attack to make the suite of tests more stable, reliable and useful for the rest of the team.

4.1 The Page Object Model

The reason to use a Page Object model is to separate the tests from the functionality on a screen. By keeping them separate, the idea is that if a page changes, or the functionality offered by a page changes, one needs only to modify the tests in one place - the definition of the page. Since the tests are consumers of a page, they do not concern themselves with the underlying implementation.

As mentioned in section three, the initial version of our automated tests did not use the page object model. When changes were made to the UI, there would be multiple tests to update. Sometimes, a test would not get the proper update and would fail accordingly. Fix that test, try again, and repeat until your fingers fall off. Contrast that prior approach with the one used in the initial Selenium based approach that uses the page object model. The tests themselves would remain untouched. Only the code that is used to represent the page that was modified gets changed. The modification should be invisible to the tests themselves. By scoping the changes only to the page object, the scope of the changes needed to match UI modifications can be kept to a minimum.

Page Objects can be used to help minimize the pain normally associated with code maintenance. If our current test automation did not use Page Objects, I probably would not have volunteered to examine the test suite.

4.2 Verify Assumptions

UI tests are fickle in that there is an inherent assumption in place that a manual tester will gloss over, but is very important to an automated test - Are you on the page that you think you are? As a manual tester of a web site, you know by looking at the browser whether or not you are at the right place. If the app is displaying the wrong page, it is easy for a manual tester to make the needed adjustments to get to the right place.

For an automated test, this usually is not so simple. The automation will be expecting that you are on the login page so that it can enter your name and password and click on the login button. But what if the test is already logged in? It will never find the login page.

One trick that I use is to verify the assumptions being used by the test.

Are you on the page that you think you are on?

Is the thing that you need to interact with being displayed?

Is it even on the page?

The answers to these questions will go a long way in predicting whether or not the tests will be reliable.

4.3 Generous Logging

One of the first things that I did to the code was to add more logging. In this day and age where disk space is cheap, it makes sense to add logging wherever possible so that any forensic analysis of test runs will contain all the information needed to be useful for troubleshooting and debugging.

Adding log messages on each transition will give more information about where on the site the test was during the run. These log messages are not super valuable when the tests are passing, but when they fail, they can give a lot of insight to what was going on, and save you time in the long run.

4.4 Unique Locators

Locators are ways to identify specific UI elements on a page. The most common way to find items are by using the HTML name, id, or class attributes. A mix of XPath and CSS selectors were used on this project, but what was troubling was the reliance on using index numbers on some selectors.

Using index numbers is not generally recommended as they are static. If changes are made to the page, the layout might also change, which will result in index numbers being regenerated. A test that is relying on a particular order of UI elements is inherently unstable and will most likely fail over time. This will lead to extra maintenance overhead as compared to tests that use a more descriptive locators.

```
// Find an element by attribute
driver.FindElement(By.Name("myName"));
driver.FindElement(By.Id("myId"));
driver.FindElement(By.ClassName("myClass"));
```

Another advantage of using good locators is that they can be immune to changes in layout. If the tests are looking for a textbox called ‘Name’ and a textbox called ‘Password’, then moving those fields will not break the tests as long as the names of those fields are not changed. The textboxes can even be moved inside a div that is displayed/hidden based on some JavaScript. As long as the names of the elements are not changed there is a good chance that changes on the UI will not affect the running of the automated tests.

4.5 Polling Sleeps

The most glaring offender in our test automation was the use of hard-coded sleeps. Our application under test relies on external jobs being run to completion before continuing. The original solution to this problem was to hard code a number to pause the tests in the hope that the jobs would finish before the test would continue. While this probably worked well on the original developer’s machine, it proved to be problematic in getting the tests to be reliable under different conditions. The solution I implemented makes use of polling sleeps.

The idea behind a polling sleep is that you would check for some expected condition. If the condition was not satisfied, the tests should sleep for a small amount of time, then recheck. This cycle would be repeated until either the condition is satisfied, or some pre-determined number of retries has been met.

Selenium has some built-in ways that this can be handled.

4.5.1 WebDriverWait

By using WebDriverWaits, there is no need to build your own wait code. You can initialize a wait object with a number of seconds to "wait" until the condition is met. One trick that I use is to ensure that the element that I would like to interact with, is actually on the page. Using WebDriverWait in this method will prevent the "No such element" exception that is rather common in automated UI testing.

```
// Create a new WebDriverWait Object with a 30 second timeout.
WebDriverWait wait = new WebDriverWait(driver, TimeSpan.FromSeconds(30));

// Wait up to 30 seconds for the element with the given Id to appear.
// Throw an exception if it does not appear within 30 seconds.
IWebElement myDynamicElement = wait.Until<IWebElement>((d) =>
    {
        return d.FindElement(By.Id("myButtonId"));
    });
```

4.5.2 Expected Conditions

Selenium takes the idea of waiting for an expected condition a step further. It exposes convenience methods that will wait for common things to occur before continuing. The actual conditions may vary slightly based on the language being used for the Selenium based tests, but the C# and Java implementations of ExpectedConditions are the richest.

The C# binding includes support for such ExpectedConditions as:

```
TitleIs
TitleContains
ElementExists
ElementIsVisible
FrameToBeAvailableAndSwitchToIt
```

Using WebDriverWait in conjunction with ExpectedConditions can save time when coding, and make the resulting tests easier to read.

The references section includes a link to the C# source code for ExpectedConditions. It should be noted that not all languages support the same ExpectedConditions.

4.6 Popup Handling

Our site makes use of a number of different kinds of popup windows — JavaScript confirm messages, alert boxes, windows that surface new functionality, and popup windows that open other popup windows.

The original solution made use of hard-coded sleeps and then switching to a window expecting that it was there. The sleeps were sufficient for most of the windows to appear, but in the failure cases - when a pop-up was not seen, or an unexpected window appeared, the tests would simply stop and log a failure.

The new solution made use of polling waits in combination with comparing window handles against a list of known windows. At a minimum, we can keep track of the main window that gets launched with the automation. By getting the list of current window handles, we can compare it to the original to find out if the current window is the main window or a popup that we need to handle.

We also need to be aware (and handle) pop-up window exceptions that caused so many problems with the original implementation. Extra care should be used when interacting with apps that use pop-up windows.

Similarly, sites that use faux pop-ups like controls that are displayed or hidden based on some condition being met can be thought of in the same light. If you need to interact with an element that is not normally displayed by default, verify that your assumptions are being met before trying to interact with those elements or else you might run into the dreaded “No such element” exception, or even worse, the “Stale reference” exception.

4.7 Frame Navigation

Most of our site does not use frames, but if you are testing a site that uses frames, just keep in mind that it’s the same as any other Selenium interaction, once you have set your browser reference correctly. Just note that you need to switch the driver reference to the level where the frame resides.

```
//Find the frame, then switch to it
IWebElement mainFrame = driver.FindElement(By.Name("MainFrame"));
driver.SwitchTo().Frame(mainFrame);
```

4.8 IE Considerations

When dealing with Selenium, there were a few issues that tripped me up when dealing with the IE browser. The most glaring was that sometimes clicking on an element had no effect for the application under test. It turns out that this is a bug with the way that the IE Driver is implemented in Selenium. This bug affects multiple versions of IE. There is a workaround — hit the Enter key while focused on an element rather than sending it a click event.

Instead of

```
button.Click();
```

Use

```
button.SendKeys(Keys.Enter);
```

5 Results

The culmination of these above changes described earlier had a great impact on the automated GUI tests. What was once a finicky test suite prone to multiple random failures became a set of tests that are more reliable. Failures reported by the test are pointing to real failures within the system under test. The logs that are created by the tests can pinpoint to a problem with the software itself or a failure on a related system (and outside the scope of the testing focus for this team).

Another big win was in the amount of time to get thru one test run. Previously, it took over 4 hours to run thru a single pass of the tests, with multiple days spent tracking down the issues. As a result, the automation could be run only once a month before a release went out. A full week was blocked out for the purposes of regression testing.

Currently, run time is down to less than 2 hours. The tests are running reliably and are now a part of nightly regression testing. Any errors being reported by the tests usually are vetted within 30 minutes. Many times, the vetting is a simple blink test based on the report of tests pass/failed. No extra time is specifically blocked out for additional regression testing. It has just become a part of the process that we go thru in testing our software.

6 Conclusion

Test automation will only be used if it can be relied upon. Presented are some general ideas of how to structure your test automation in a way that can be easily maintained in the future. This paper also presented some ideas specific to using Selenium. While the code used in this paper is based on the C# bindings, the ideas can be translated to the other bindings, as well as to other GUI automation frameworks as well. In fact, many of the ideas used on this project come from my work in Ruby and Web Application Testing in Ruby (Watir).

Web sites are becoming increasingly "responsive" in their behavior, which introduces challenges when trying to validate functionality from the GUI perspective. By implementing robust test code, the test suite will be better suited to handle the irregularities that abound when dealing with communications on the web. This greater reliability will be a key factor to whether or not your automation framework will be used, or become dreaded shelfware.

Avoid shelfware and your organization will be better for it.

References

SeleniumHQ. "Selenium WebDriver." http://docs.seleniumhq.org/projects/webdriver/ (accessed June 26, 2015).

World Wide Web Consortium. "WebDriver" https://w3c.github.io/webdriver/webdriver-spec.html (accessed June 26, 2015)

Selenium. "Page Objects" https://code.google.com/p/selenium/wiki/PageObjects (accessed June 26, 2015)

GitHub. "Expected Conditions" https://github.com/SeleniumHQ/selenium/blob/master/dotnet/src/support/UI/ExpectedConditions.cs (accessed June 26, 2015)

AutoIt. "AutoIt". https://www.autoitscript.com/site/autoit/ (accessed June 26, 2015)

Watir. "Watir". http://watir.com/ (accessed June 26, 2015)