Obtaining True UI Automation Speed

Sam Woods

sam.woods@puppetlabs.com

Abstract

How many people get frustrated after spending hours writing tests, then even more time debugging failures before finally being able to determine a root cause or fix? I have spent a lot of time thinking about this and experimenting with ways to increase efficiency for these 2 aspects of UI automation. Through trial and error throughout my career, I have concluded that there are simple and effective ways to increase velocity of test authoring and root cause analysis of test failures, which are often overlooked. Even teams with relatively mature automation processes can be blissfully unaware of the opportunities they have to make major improvements to their automation velocity.

Besides the well known Page Object Pattern [1], we will walk through effectively using setup and cleanup functions, how to properly structure helper functions, how to model data consumed by the application, and more. Finally, techniques for increasing the speed of determining the root cause of automation failures will be demonstrated, such as: built in logging, taking screen shots on failure, useful integration with your defect tracking tool, and determining which failures are likely to be the same root cause through stack trace analysis. By including these relatively simple strategies in your own automation, you could increase your efficiency and be able to add more automated tests while spending less time debugging failures.

Biography

Sam Woods recently joined Puppet Labs as a Senior Software QA Engineer with 17 years of QA and automation experience, including 13 years at Microsoft. He is a self taught developer who has evolved to the point of helping multiple mid to large size organizations create automation strategies, build strong automation teams, create the necessary automation infrastructure to support the strategies, and then execute on them. He has presented on UI automation at the local Portland Selenium meet up and presented multiple times to audiences as large as 300 people within the companies he has worked for. He currently has the 5th highest all time ranking on the SQA Stack Exchange [2] forum. Sam gets excited about improving efficiency through both automation and best practices, as well as getting into the weeds with automation and automation infrastructure. Outside of work, Sam enjoys singing karaoke and spending lots of daddy time with his two young daughters.

1. Introduction

Are you working on, or have you ever worked on automation that just seemed un-maintainable? Automation you spent more time diagnosing and fixing problems with than you spent adding new test cases? Have you had automation that was so unreliable that you couldn't trust the results even when tests passed, or had large portions of the automated tests failing nearly every run, seemingly at random? It's enough to make even the most stoic automation engineer pull their hair out in frustration.

Many have attempted to find solutions and failed, and many ignore the problems and continue writing new tests while amassing additional technical debt. The title of this paper refers to the fact that many teams writing UI are focusing on solving the wrong problems, and for example care more about the execution speed than test authoring speed. But, don't fear! While these symptoms can be daunting, they are often simpler to find and fix than you might imagine.

I will provide solutions and examples that will help you increase the speed of writing automated tests as well as decreasing the time to determine the root cause of failures (the two areas I refer to with 'true UI automation speed')

2. The Problems

The first step is a diagnosis. What exactly is the root cause of these symptoms? The 3 most common problems that I have observed that result in the symptoms stated above are:

- Instability in the product itself
- Too many automated tests
- Unreliable automation

Typically, you will see some combination of the 3 problems listed above. I will outline these problems below, but the primary content of this paper will be directed towards the third issue, unreliable automation.

2.1 Instability in the product itself

- Constant and major changes to various portions of the user interface
- New features that required rework of old features.
- Major updates to or overhauls of existing features.

If there is a lot of instability in your product - especially if you are working in an agile environment - your team should ask the following questions of themselves:

- When we implement features, are they really "done" done?
- Are your customers actually happy with or benefiting from these changes? Or perhaps they are just as frustrated as the QA staff that things keep changing out from under them?

If you answered yes (and really feel like that's an honest answer) to both of these questions, I would challenge you to gather some actual data on the above points and review it.

2.2 Too many automated tests

- A large automation code base with many hundreds or thousands of tests.
- Automated tests written years ago that aren't updated unless they break.
- No clear idea of what kind of test coverage you get by executing the tests.
- Lots of duplicated or largely equivalent tests.

Often times QA teams will horde tests for no apparent reason. Perhaps out of nostalgia, or even fear? If you are suffering from an overwhelming pile of unmaintained automated tests, I would pose the following questions:

- What value are you getting out of your automated tests?
    - Is executing these tests allowing you to skip manual testing for entire features?
    - Are they an immediate and reliable indicator of the quality of your application?
    - Is every test validating a unique aspect of the application or feature?
    - Can you easily determine the answers to the above questions?
- How long is the feedback loop?
    - Is running 5000 tests (or insert your own number of tests here) even manageable when there are failures to investigate?
    - Can you provide valuable feedback on a build within an hour (or even 5-10 minutes) of when new code is checked in and deployed? (Don't forget to include how long it takes you to triage failed test results)

If the answer to any of the above questions was no, then you need to face your fear of cutting out the crap, wade in and start shoveling. Take a rational approach to trimming down the volume of tests. Which tests are virtually equivalent and can be consolidated into a single test or just removed (see section 4.3 about using code coverage results as one avenue to determine this)? Which tests cover the highest priority or most used customer features? Which test failures would be the highest severity and have the most impact to customers? These tests should be prioritized, and the rest either de-prioritized, consolidated, or removed.

Do, however, take caution in your diagnosis. I have seen teams with a couple hundred tests come to the same conclusion as teams with a few thousand tests, and while you may have some unnecessary or inefficient tests, the primary issue may just be…

2.3 Unreliable automation

- Test results that are unclear, incorrect, or not reproducible.
- Tests that fail when they should pass, or vice versa.
- Flaky tests that return seemingly random results.
- Tests that are brittle and break due to even minor changes in the application.
- Tests that pass when the site is running fast, but not when it takes a couple seconds longer to respond.

I will be detailing ways to increase the speed of automated test authoring and troubleshooting, while simultaneously improving reliability and reducing the overall cost of maintenance for the remainder of this paper. All examples in this document are pseudo-code and are not in any particular language.

3. Solutions

We'll start with some strategies to improve the velocity of test case development, and overall quality and maintainability of your automation:

3.1 The Page Object Pattern

Many people believe the page object pattern is synonymous with Selenium WebDriver's [3] implementation of it, and I think that is a mistake. I personally dislike the PageFactory, and would much rather see lazy initialization of the web elements allowing you to simply create a page class and create instances of all of the element objects right in the constructor. The concept of the page object pattern is simple, and it is extremely important when it comes to increasing efficiency of writing and maintainability of automated tests. It is the minimum bar for increasing maintainability of automation.

Don't over-complicate it. The basic idea of the Page Object Pattern is that you want to separate your test cases from the definitions of your pages. That's it. You can use whatever implementation you're comfortable with, as long as you're doing that.

Bad Example:

```
public void Test1(){
   FirstNameTextBox = new WebElement(…);
   FirstNameTextBox.SendKeys("John"); // Hard coding name values
   LastNameTextBox = new WebElement(…);
   LastNameTextBox.SendKeys("Doe"); // Hard coding name values
}
```

Good Example:

```
public class MyPage {
   FirstNameTextBox = new WebElement(...)
   LastNameTextBox = new WebElement(…);

   // A new helper function to enter name, avoiding hard coding values
   public void EnterName(String FirstName, String LastName){
      FirstNameTextBox.SendKeys(FirstName)
      LastNameTextBox.SendKeys(LastName)
   }
}

// Somewhere in your test class:
// You will need to initialize the page class somewhere,
// usually in a constructor, or setup method.
public void Test1(){
   myPage.EnterName("John", "Doe")
}
```

This is critical to any UI automation. Without this abstraction in the example above, if anything at all changes about entering your name on the page, fixing your automation would entail finding all instances where you are referencing those elements and updating them - assuming you are even using the same names or locators for those elements in the first place. You may have them defined multiple different times using different criteria and different names, making them all but impossible to find without executing your automation, and then triaging failures that you fix one at a time. By contrast, if you use the good example, if something about the form to enter a name changes, you update the one method in the page class and fix all of your tests at once.

3.2 Modeling User Defined Data

This is an important one. In most modern web applications there are complex data constructs at play - maybe a user profile, shopping cart, array of messages, or any other number of logically grouped data sets. If anything like this exists in your web application it needs to be modeled in your automation, just how developers model it to keep it maintainable within the application itself. Modeling it is usually not a difficult task, you look at how and where the data shows up in your web site, and you create a simple class structure to make it easy to enter and validate the data. I'll use a relatively simple one for an example:

```
public class User {
   string FirstName
   string LastName
   string Address1
   string Address2
   string State
   int ZipCode
   string PhoneNumber
   string UserName
   string Password

   // Note the use of optional parameters with default values in the
   // constructor, most languages support this.  If you're unfamiliar
   // with it, look it up for your language of choice.
   public User(firstName = "testFirst", lastName = "testLast"…){
      FirstName = firstName
      LastName = lastName
      … // Continue assigning the class properties to the method argument values.
   }
}
```

The above is very simple, but it amazes me how many times I see hard coded values all over the place in automation. The above data structure now ties in easily with and simplifies your page object helper functions as well. Imagine a single helper function called "SignUp" that takes a User object as a parameter. If a new field for Gender is added to the signup page, or a phone number removed, or any other change at all, it will be very simple for you to modify that one SignUp function and one User class and all of your existing automation will again function normally. Now let's say you add various payment methods to your site, and you want to be able to tie them to that user. Easy, create a PaymentMethod class that can be either a credit card, or Paypal, or whatever else you may accept for payment and make it a property of the User class. Maybe most Users won't have payment methods on file? No problem, make it null by default and have your helper functions ignore it if it is null.

3.3 Helper Functions

Helper functions are a simple concept, but can be tricky to get right. Basically, helper functions fall into two categories. They are functions to either simulate user behavior, or validate expected results. I have seen people create helper functions called "EnterTextIntoNameField", or "ClickSubmit" where all they are doing inside the helper function is a single call to a webdriver method or property. This may be overkill, especially if these are all part of a form with a dozen other fields on it. It may also be perfectly reasonable in other circumstances, if say you wanted to include some additional logging, or needed to do some tricky waiting for certain parts of the page to load prior to entering text, or a number of other things that can complicate a relatively simple thing like entering text or clicking a button.

Common areas of confusion:

- How much should you put into a single helper function? Can you nest smaller helper functions into one larger one?
- What should the scope of a helper function be?
- Can a helper function span multiple pages?
- Should you have separate functions for entering data and submitting data?
  - What if you need to test a specific field in a form, but want to automate entering data into all of the other fields?
  - What if you are purposely entering bad data to validate error messages?
- Should the helper function throw an exception if it doesn't complete successfully, or return a Boolean value to assert on?

- Should you log what you're attempting to do, as well as the result?
- What data, and in what format do you pass in to a helper function?
- Should you return the page object of the page the helper function transitions to?

Many of these are up to the preference of the developer, but this is what I have adopted as best practices, and have found to work well:

- Helper functions can be as narrow or deep as they need to be. If it makes sense to break a registration page up into "EnterCredentials", "EnterProflileInfo", "EnterBillingData", etc, that's fine, but also include a "Register" method that encapsulates all of those to make it easy from a test case, especially if most of the time you won't need that granularity except one off tests.
- Parameters to helper functions should typically be a single object, whether that is a string for a method such as SelectEmailByTitle(String title) or an instance of a custom class for a method such as Register(User user).
- Helper functions can definitely span multiple pages, especially in the case of multi-page forms, or navigation methods where there is no direct way to navigate to the other page.
- When entering and submitting data, have a single method to both enter and submit the data.
    - Include an optional Boolean parameter "submit" and have it default to true. If you are writing field validation tests for the form, you can now fill out most of the form, then enter specific data for the one field you care about before submitting.
    - Include a Boolean parameter "success" that defaults to true, so you can validate errors if necessary by asserting on the result of the method. If you don't expect an error, validate in the helper method that you arrived on the correct following page. If an error is expected, create a separate method to validate the error message.
- I am a fan of throwing exceptions in helper methods and having them bubble up, unless the helper method is specifically a validation helper method. Here are some reasons why:
    - Many test runners differentiate between an exception (something broke before your test even got to the point of validating what it was intended to validate) and an assertion failure (The thing the test was trying to validate was broken). This is a very useful piece of information when it comes to determining the root cause of a test failure. With this information, you will know definitively if the test itself failed, or if some step in the setup steps failed.
    - In the case of a validation helper method such as "ValidateMissingInfoMessage", you should return a Boolean value signaling success or failure, which could then be called directly from an automated test case where you assert on its response.
- My preference is having a static class with a list of page instances rather than returning the page object instance in helper functions. This is entirely preference. Your automation will be successful regardless of which approach you choose. I like the readability of a test case a bit better using a static class, but if you like having the type of page returned by the method to make it simpler to determine where you are supposed to end up, that works too.

Here is one example of what I would consider a good helper function, making use of the User class we saw in the "Modeling User Defined Data" section:

```
public class RegistrationPage{
    // All elements defined with locators
    //(CSS or XPATH definitions of the element)

    // Use our custom Users class with a default user already set up
    public RegisterNewUser(User user = Users.defaultUser,
      Boolean submit = true, Boolean success = true){
      log.info("Registering new user " +
      user.firstname + " " + user.lastname)
      try {
         if (user.firstname != null){
            firstNameTextBox.text = user.firstname
         }
         if (user.lastname != null){
            lastNameTextBox.text = user.lastname
         }
         // etc...

         If (submit){
            log.debug("Submitting registration.")
            registerButton.click()
         }
      }
      catch (exception ex) {
         // Whatever you need to do to log the caught exception details.
         log.error("User registration failed!", ex)
      }
      log.debug("Completed registration")
      if (success){
         // Make sure we're actually on the welcome page
         welcomePage.verify()
      }
   }

}
```

3.4 Effective use of Setup and Cleanup functions.

Setup and cleanup functions can save you from copying a lot of boilerplate code for similar test cases. They can also be used to decrease execution time and increase maintainability. Finally, it also makes it very clear if there is a failure in a shared setup or cleanup function that the actual functionality the test is asserting is not the culprit. This makes it simple to group common failures together into likely buckets of the same root cause, which can turn investigating 30 test failures into investigating only a handful of unique root causes. More on this later.

Because setup and cleanup functions are critical and need to be as reliable and execute as fast as possible, it is not always necessary to use the UI at all for them. Let’s say you have a test where you are associating 2 user accounts in some way, maybe making one user a child of another user. You shouldn't care about going through the UI to create two new users prior to associating them, they just need to exist so you can test the critical functionality of associating the two users in the UI. This can be accomplished by making HTTP requests directly to the web server. If your team is already using a tool for performance testing, you likely already have most of what you need to construct these requests. If not, and you are starting from scratch, there is a little more to it, but it is worth the effort to invest in automating the web requests directly. It is even possible to send an HTTP request to authenticate, use Selenium (or whatever automation tool you are using) to set cookie values for authentication and then navigate directly to a page on your site, already logged in – without even going through the UI at all!

Whether you make use of HTTP requests in the setup and cleanup functions or not, here are the best practices I have adopted in terms of how and when to use project, class and test level setup and cleanup functions.

- Project level setup and cleanup

    - The project setup method should setup anything that will be used in all tests. For example, it may setup and start a logger, or create a Selenium driver instance. It could also initialize any shared objects such as default users, or create instances of all of your page classes, etc.
        - One note about Selenium driver instances – if you want to execute your tests in parallel and are using a Selenium Grid, or cloud based Selenium Grid such as Sauce Labs [4], you may have to quit and create a new driver instance for each test, in which case those should live in the test case setup and cleanup.
    - Project cleanup similarly can be used to quit selenium, stop loggers and summarize logs, destroy objects in memory, etc.
    - In the case where the test execution framework does not have a project level setup and cleanup, I commonly have a base test class that I inherit from and then execute the setup and cleanup from the super class. Different test runners handle inheritance differently, so refer to your documentation or experiment a bit to ensure it is working the way you intend it to.
- Class level setup and cleanup
    - Group test cases together by feature or functionality into buckets of similar tests. For example, you may have a class of tests that simulate users making purchases in a shopping cart and because every test needs a user to log in as. You can create 1 user in the class setup and use that user in all of your tests.
    - The cleanup may delete the user, or purge the purchases from the database.
- Test setup and cleanup
    - Often, you should have a test method in a base test class that logs into the application and does any other boilerplate setup that may be necessary.
    - The test setup is also where you can include common steps for the particular class of tests. In the example of a shopping cart, your test setup could log in and navigate to a product search page prior to every test beginning.
    - Avoid using the test cleanup function to get back into a good state for the next test case to run, there are too many potential problems with that approach. You are nearly always better off starting over for every test with a fresh new driver instance.

I will not provide a full example for this topic, simply because it would require showing the skeletons of many multiple related classes, but I will speak a bit about why this is important to do well and share a small example with basic usage. Imagine an e-commerce site that allows you to search for and then purchase products. A few classifications of tests for that site could be:

- Login and registration
- Product search
- Product details
- Comments and ratings
- Purchases

Each one of these classes of tests have very different pre-requisites. Some may not require being logged in to the site, but purchasing or commenting/rating likely would require that. In order to make a purchase, there are likely steps that you need to take such as searching for and selecting products and entering or validating billing and shipping information.

Here is an example of a test class containing tests for purchasing products:

```
public class PurchaseTests inherits BaseTest {
    ClassSetup() {
       Users users = new Users()
    }

    TestSetup() {
       Driver driver = new Driver(...)  // Create the instance of the browser driver
       LoginPage.login(users.default)
       MainPage.search('laptops')
       SearchResults.selectResult(0)
    }

    TestCleanup() {
       If (!testResult) {
          // Take screenshot on failure
       }
       Driver.quit() //  Clean up the instance of the driver
    }

    Test1(){
       // Some test content
    }

    TestN(){
       // Some test content
    }
}
```

In this relatively simple example, without the use of setup and cleanup functions you would be copying and pasting about 8 lines of code for every single test case, and of course if anything were to change in those 8 lines of code, you would be looking at a large search and replace task. Make it easy for yourself, use setup and cleanup functions wisely.

3.5 Reducing the noise

Finally, in addition to everything else, there are a few things you can do to simply reduce the likelihood of your tests failing in the first place.

- Linking dependant test results. If the login test fails, there are probably a lot of other tests that will fail as well. You can create dependencies either within the unit testing framework, or simply within your reporting to tell you when a test's results should be ignored because it was really a failure due to the dependent test, and not the test itself.
- Making element discovery less brittle. You can ask developers to add ID's to elements and NOT change them. You can use CSS or XPath selectors that do not contain the entire path to the element (do you care if a div way up at the top of the DOM tree changes?) Have it be as concise as possible. For example, a CSS selector like this one: "table#messages > tr > td > div > ul > li > a[href='foo']" could be condensed to '#messages a[href='foo']". The smaller more concise selector is much easier to maintain, because the structure of the table and it's descendants could change, and your selector will still find what it is supposed to.
- Making element interaction less brittle. You should be using implicit waits everywhere. Implicit waits are built in to most UI automation tools, but an implicit wait basically will always try and retry any action on an element until successful, or until a timeout occurs. Explicit waits on the other hand you must call directly in your code. Set the time to wait for elements to something reasonable, and ALWAYS wait for the element to be ready first. If you are running into Stale Element exceptions, make sure to wrap those in a try/catch and retry.

4. Logging and Reporting

4.1 Logging

I admit, I may be a little bit crazy when it comes to logging. My goal with logging is that I never have to debug an automation failure ever to figure out why it failed. I provide enough information to be able to peruse the log, and know within a few seconds exactly what went wrong. Whenever a test fails and there *isn't* enough data in the logs, part of me fixing the test is adding logging to make it easier to diagnose the next time there is a failure. This may seem like a lot of overhead, but when done right, it actually saves a dramatic amount of time in diagnosing failed tests, and makes all of the code more readable and understandable.

What should you log? You should have logging primarily at 3 different levels.

- Test execution level
  - Examples:

```
------------------ Project setup beginning ------------------
------------------ Class setup beginning for class myTestClass ------------------
------------------ Test setup beginning ------------------
------------------ Test case abc123 beginning ------------------
```

- Test case level (usually not very much, if any logging here)
  - Examples:

```
Step 1: Log In
The actual message "ERROR" did not match the expected message "SUCCESS"
```

- Page object level
  - I log the attempt and outcome of every single helper method, and often actions within the helper method. Remember the logging included in the example helper function?
  - Examples:

```
Attempting to register user TestFirst TestLast
Submitting registration
Registration completed
An unexpected exception occurred attempting to submit the form.  Exception message:
'message'
```

- Logging at the element level
  - Examples

```
Entering text 'TestFirst' into field with locator 'input#first-name'
Clicking element with locator 'button.submit'
Unable to find element with locator 'select.choose'
```

When you combine all of this logging, you get very specific details of what was happening in your test when a failure occurred, and combined with the exception and stack trace information, you should be in a lot better shape to determine the root cause of failure.

Logging at the element level for Selenium requires creating an event listener and listening for events. Other frameworks have their own preferred methods for doing so. If doing this yourself sounds daunting, here is a simple write up of how to do so for Selenium: http://elementalselenium.com/tips/55-wrapper [5].

Another important logging tool for UI automation is screenshots on failure. These can be infinitely helpful when you come up against that one inexplicably flaky test that doesn't fail when you run locally, or even on retry of a test. Looking at the screenshot is usually the first place I go after the log if I still need more information. Most frameworks have a built in function for taking screenshots, although depending on which test execution framework you are using, it may be trivial or tricky to determine the test result in a cleanup method where you can execute the test. Refer to your test execution framework documentation for details.

4.2 Reporting

Most unit testing frameworks produce a report for an individual execution of tests. This is the bare minimum you need to triage test failures for that particular run. Having historical test result data can make prioritizing and triaging test results much more efficient. Typically I have a way to store results only for "official" test runs, so you don't get your database filled with results from you debugging or creating tests locally. You can create reports to help you determine where you should be spending your time. Some interesting reports are:

- Your least reliable tests (change most frequently between pass/fail) over the last 10 runs – These should be given more attention. You can also group the failures by stack trace (see explanation below). If the top stack trace is the same, you know they are failing for the same reason each time.
- Status of the most recently added tests – New tests that fail soon after being added, should be looked at immediately.
- Tests that have had the most defects associated with them – This usually indicates an area of the application that is unstable, or likely to have failures. You can focus more effort on adding additional automated tests in these areas of the application.

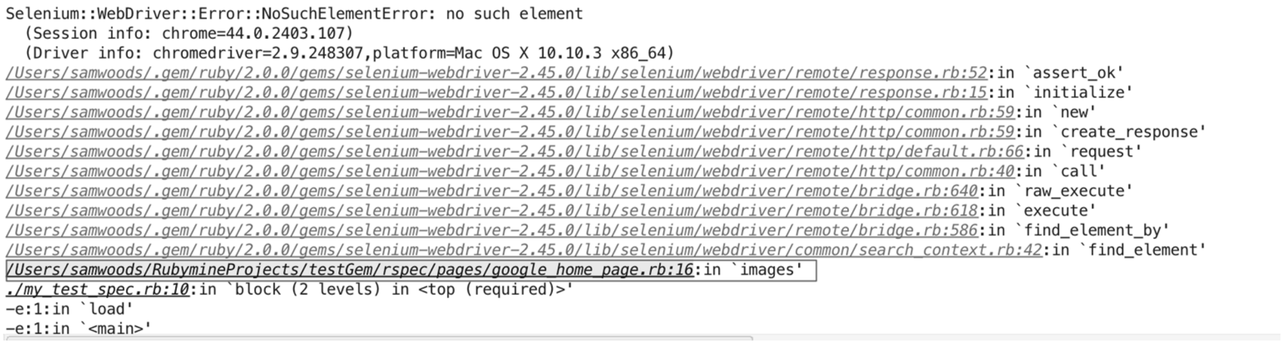
Most unit testing frameworks support about 3 different result types: Pass, Fail and Ignore. I usually want more than that. One that I find extremely valuable to track is "KnownFailure". Basically, when a test fails due to a known defect, you want to know that it failed, but you don't need to waste your time triaging that test build over build, or include that failure as part of a quality gate that would prevent the build from being moved forward. I have extended 3 different Unit testing frameworks (Nunit [6], Junit [7] and Spock [8]) to support this. Usually you can annotate a test case method like this:

```
@KnownFailure(D12345)
public void MyTestCase(){
…
}
```

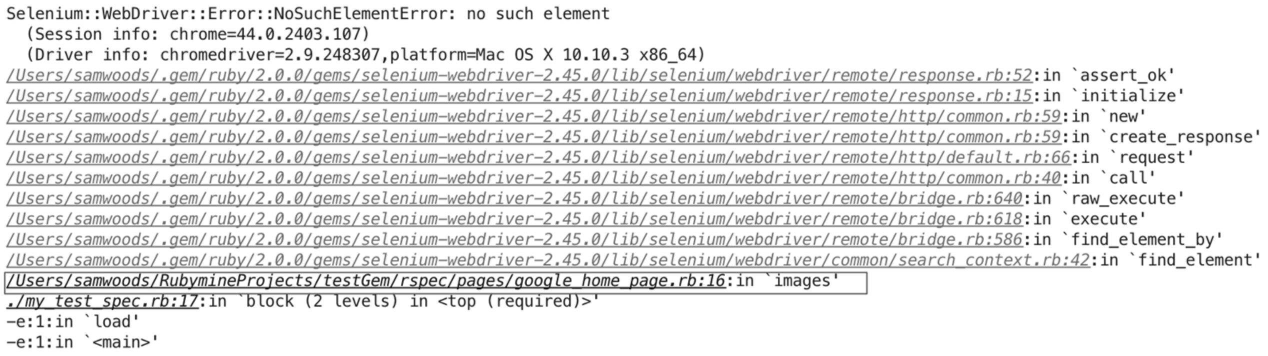
D12345 would be the defect ID, and my extension will be smart enough to query the defect tracking tool for the status of the defect. If the defect is not fixed, I would ignore running the test. If it has been fixed, I would run the test as normal. Depending on your defect tracking tool, you may be able to use other information as well, such as whether the defect is "ready to test", or even look at the "fixed in" build number to determine whether the build you are testing contains the fix. You may even want the result of the automated test to automatically move the defect into an accepted or rejected state, although I would be cautious with that.

When I store data in a reporting database, I always like to include the stack trace. One awesome thing you can do is group failures by the top stack trace. By top stack trace, I mean that you choose where to cut off the stack, usually you can ignore anything that happens in the test case code or prior to the page object code, and save only the trace from your page object helper and later. These failures are very likely to have the exact same root cause, meaning if you have a test run where 20 tests fail, but they are bucketed into only 2 groups of top stack traces, then you only need to investigate 2 failures not 20. The idea is pretty simple, take your stack trace and draw a line in between the execution of a method from your test case and the execution of a method from your page object and throw out everything from the test case and up. Then draw a line between the last execution of a method from your test execution framework and your page object and throw out everything afterwards. What you are left with is your "top stack trace". You can use namespaces and other groupings in your code to make this easier to do in an automated way. If entering data into a form on a registration page is failing for some reason and you have a helper function in your page class for filling out the form, then if you remove the test case specific code from the stack trace, and the test framework code and the remaining stack trace should be the same for any test case that is executed and failing to fill out the form correctly.

Here is a simple example automating google search [15] with Selenium in Ruby[9]:



```
Selenium::WebDriver::Error::NoSuchElementError: no such element
  (Session info: chrome=44.0.2403.107)
  (Driver info: chromedriver=2.9.248307,platform=Mac OS X 10.10.3 x86_64)
/Users/samwoods/.gem/ruby/2.0.0/gems/selenium-webdriver-2.45.0/lib/selenium/webdriver/remote/response.rb:52:in `assert_ok'
/Users/samwoods/.gem/ruby/2.0.0/gems/selenium-webdriver-2.45.0/lib/selenium/webdriver/remote/response.rb:15:in `initialize'
/Users/samwoods/.gem/ruby/2.0.0/gems/selenium-webdriver-2.45.0/lib/selenium/webdriver/remote/http/common.rb:59:in `new'
/Users/samwoods/.gem/ruby/2.0.0/gems/selenium-webdriver-2.45.0/lib/selenium/webdriver/remote/http/common.rb:59:in `create_response'
/Users/samwoods/.gem/ruby/2.0.0/gems/selenium-webdriver-2.45.0/lib/selenium/webdriver/remote/http/default.rb:66:in `request'
/Users/samwoods/.gem/ruby/2.0.0/gems/selenium-webdriver-2.45.0/lib/selenium/webdriver/remote/http/common.rb:40:in `call'
/Users/samwoods/.gem/ruby/2.0.0/gems/selenium-webdriver-2.45.0/lib/selenium/webdriver/remote/bridge.rb:640:in `raw_execute'
/Users/samwoods/.gem/ruby/2.0.0/gems/selenium-webdriver-2.45.0/lib/selenium/webdriver/remote/bridge.rb:618:in `execute'
/Users/samwoods/.gem/ruby/2.0.0/gems/selenium-webdriver-2.45.0/lib/selenium/webdriver/remote/bridge.rb:586:in `find_element_by'
/Users/samwoods/.gem/ruby/2.0.0/gems/selenium-webdriver-2.45.0/lib/selenium/webdriver/common/search_context.rb:42:in `find_element'
/Users/samwoods/RubymineProjects/testGem/rspec/pages/google_home_page.rb:16:in `images'
./my_test_spec.rb:10:in `block (2 levels) in <top (required)>'
-e:1:in `load'
-e:1:in `<main>'
```



```
Selenium::WebDriver::Error::NoSuchElementError: no such element
  (Session info: chrome=44.0.2403.107)
  (Driver info: chromedriver=2.9.248307,platform=Mac OS X 10.10.3 x86_64)
/Users/samwoods/.gem/ruby/2.0.0/gems/selenium-webdriver-2.45.0/lib/selenium/webdriver/remote/response.rb:52:in `assert_ok'
/Users/samwoods/.gem/ruby/2.0.0/gems/selenium-webdriver-2.45.0/lib/selenium/webdriver/remote/response.rb:15:in `initialize'
/Users/samwoods/.gem/ruby/2.0.0/gems/selenium-webdriver-2.45.0/lib/selenium/webdriver/remote/http/common.rb:59:in `new'
/Users/samwoods/.gem/ruby/2.0.0/gems/selenium-webdriver-2.45.0/lib/selenium/webdriver/remote/http/common.rb:59:in `create_response'
/Users/samwoods/.gem/ruby/2.0.0/gems/selenium-webdriver-2.45.0/lib/selenium/webdriver/remote/http/default.rb:66:in `request'
/Users/samwoods/.gem/ruby/2.0.0/gems/selenium-webdriver-2.45.0/lib/selenium/webdriver/remote/http/common.rb:40:in `call'
/Users/samwoods/.gem/ruby/2.0.0/gems/selenium-webdriver-2.45.0/lib/selenium/webdriver/remote/bridge.rb:640:in `raw_execute'
/Users/samwoods/.gem/ruby/2.0.0/gems/selenium-webdriver-2.45.0/lib/selenium/webdriver/remote/bridge.rb:618:in `execute'
/Users/samwoods/.gem/ruby/2.0.0/gems/selenium-webdriver-2.45.0/lib/selenium/webdriver/remote/bridge.rb:586:in `find_element_by'
/Users/samwoods/.gem/ruby/2.0.0/gems/selenium-webdriver-2.45.0/lib/selenium/webdriver/common/search_context.rb:42:in `find_element'
/Users/samwoods/RubymineProjects/testGem/rspec/pages/google_home_page.rb:16:in `images'
./my_test_spec.rb:17:in `block (2 levels) in <top (required)>'
-e:1:in `load'
-e:1:in `<main>'
```

You can see that the stack traces are nearly identical, the only real difference is the line number in the test_spec file. Two different test cases that both called the same helper function, and both failed in different places. It really is that easy, in fact the only line you care about from this particular trace is the line coming from the google_home_page.rb file. In more complex scenarios, there may be many lines you care about, primarily if you have helper functions calling other helper functions within the same page, or in the base page. Once you pull out that section, and determine it is identical between failures, there is a very good chance that all of those failures where the stack trace matches are due to the same root cause, and youre investigation time is dramatically decreased.

4.3 Code Coverage against functional tests

Code coverage can be a bit tricky to set up for functional tests. Historically you have needed an instrumented build, which is pretty easy when executing unit tests in the build pipeline, but a little trickier to deploy to a full test environment. These days there are a number of code coverage tools that you simply set up agents on the SUT, and point it to the source. Some examples are Clover [10] for Java [11] and NCover [12] for .NET [13]. Many web applications are also a lot heavier on the front end JavaScript [14], making JavaScript code coverage more relevant and important as well. Ideally, the code coverage tool would support storing results per test case for reasons outlined below.

Code coverage on functional tests can be useful for the following reasons:

- Find areas of the code that are not covered that should be, and add automated tests.

- Find tests that overlap, or are equivalent. This is where storing results on a per test basis is handy, so you can compare coverage between tests. If you have test cases that do not execute any unique paths in the code, they are basically equivalent to other tests and are probably adding very little value and can be removed. The less tests you have, the easier they are to maintain and triage.
- A combined coverage with unit tests and functional tests, so you can work with developers to see what code paths should be covered, or are easier to cover with unit tests VS functional tests.

5. Conclusion

These tips all come from years of trial error and experimenting on my part. I hope that you have learned something new, or have been prompted to think about some new possibilities. If it all seems overwhelming, pick just a couple of things to try to start with. I am certain they will improve your automation and simplify the process of writing tests and investigating failures, just as they have for me.

I have identified a lot of low hanging fruit for you, such as implementing the Page Object Pattern and using effective Helper methods with proper data models, Some of the other recommendations like automating the process of checking your defect tracking tool, setting up code coverage, or setting up a reporting framework and building some reports may take a higher degree of effort, but I am confident that all of them will add value if the investment is made. Whatever you do, don't continue to build up technical debt! Set aside some time to implement some of the ideas I have shared, or that you have found elsewhere.

Automation code should be given the same level of scrutiny as the code it is testing. A failure in the automation that allows regressions through can be just as damaging as a failure in the system under test, so start treating it that way and watch how quickly it can transform into a lean, fast, powerful and absolutely invaluable tool for you and your team. Happy automating!

6. References

[1] Page Object Pattern, https://code.google.com/p/selenium/wiki/PageObjects (accessed August 1, 2015)

[2] SQA Stack Exchange. “User Reputation Leagues - Software Quality Assurance & Testing - All Time - Stack Exchange” http://stackexchange.com/leagues/111/alltime/sqa (accessed July 29, 2015)[3] Selenium Webdriver, http://en.wikipedia.org/wiki/Selenium_(software) (accessed August 1, 2015)

[4] Sauce Labs, https://saucelabs.com/ (accessed August 1, 2015)

[5] Haeffner, Dave. 2015. “How To Add A Wrapper To Your Selenium Tests”. Elemental Selenium. http://elementalselenium.com/tips/55-wrapper (accessed July 29, 2015)

[6] NUnit, https://en.wikipedia.org/wiki/NUnit (accessed August 1, 2015)

[7] JUnit, https://en.wikipedia.org/wiki/JUnit (accessed August 1, 2015)

[8] Spock, https://en.wikipedia.org/wiki/Spock_(testing_framework) (accessed August 1, 2015)

[9] Ruby, https://en.wikipedia.org/wiki/Ruby_(programming_language) (accessed August 1, 2015)

[10] Clover, https://www.atlassian.com/software/clover/overview (accessed August 1, 2015)

[11] Java, https://en.wikipedia.org/wiki/Java_(programming_language) (accessed August 1, 2015)

[12] NCover, https://en.wikipedia.org/wiki/NCover (accessed August 1, 2015)

[13] .NET, https://en.wikipedia.org/wiki/.NET_Framework (accessed August 1, 2015)

[14] JavaScript, https://en.wikipedia.org/wiki/JavaScript (accessed August 1, 2015)

[15] Google Search, https://en.wikipedia.org/wiki/Google_Search (accessed August 1, 2015)