

An Introduction to Test Automation Design

Copyright Lisa Crispin 2010

Reprinted from *The Testing Planet*, July 2010, No: 2 www.softwaretestingclub.com

There's no shortage of excellent test automation tools available today. Many are open source, so the up-front cost is low. Some come with script capture tools or mini-IDEs that speed the learning curve. Many have excellent user guides, tutorials and screencasts available to help testers and teams learn them.

The Problem

One would think that with all this tools and help available, test automation efforts would be more likely to succeed than in years past. However, what I see all too often are teams who had no trouble creating thousands of automated regression tests – the problem comes a few months down the road, when they spend most of their time maintaining the automated tests. These testers are left with very little time with critical testing activities such as exploratory testing or collaborating with customers up-front to get examples of desired software behavior. Now they're between a rock and a hard place: regression testing manually takes too long, and so does keeping the automated tests up to date as the production code changes.

In my experience, this quandary is usually the result of poor automated test design. Automated test code is, well, code. It requires the same thoughtful design as production code. By applying coding principles such as "Don't Repeat Yourself", anytime we need to update a test to accommodate production code changes, we only have to change one module, class, include or macro, perhaps only a variable value.

Unfortunately, test automation is often done by testers who don't have much, if any, programming experience. They may be capable of capturing scripts with a tool and replaying them, but this is only a way to learn a tool – not a way to design real tests. The programmers on their projects may feel their job is to write production code, not help with functional or GUI test automation. So, the testers do their best, and end up with half a million lines of captured scripts (yes, I know of more than one example) that are impossible to maintain.

Even if teams use a framework that allows non-programming members to write tests in plain English, a spreadsheet or some other non-programming language, the test cases still need proper design. I personally have made a terrible mess automating dozens of FitNesse tests where I repeated the same steps over and over, because I didn't know at the time about the `!include` feature of FitNesse.

Possible Solutions

A simple solution is for experienced programmers to pair with testers to automate tests. At minimum, teams need to hire testers with solid code design skills to help the non-programmer testers. Testers need training in basic design principles and patterns.

There are lots of resources to learn how to do a particular type of automation, such as GUI test automation. My personal mission is to find ways to help educate testers in designing maintainable tests. As a start, I'd like to illustrate some basic design principles using

examples written in Robot Framework with Selenium driving GUI tests. I had not used either Robot Framework or Selenium before when I sat down to create these examples. And while I've been automating tests for a long time, I'm not too good with object-oriented programming. If you're experienced with these tools, you may find lots of faults with my test code. But I wanted to demonstrate that these design principles work even when you're unfamiliar with your tool or scripting language.

If you want to play with similar examples, you can download the following:

Information and downloads to install Robot Framework:

<http://code.google.com/p/robotframework/>

Robot Framework Selenium Library downloads, including the demo:

<http://code.google.com/p/robotframework-seleniumlibrary/>

Look up Selenium commands here <http://code.google.com/p/robotframework-seleniumlibrary/wiki/LibraryDocumentation>

Run tests with ".\rundemo.py <test file name>"

To start the server for the app under test manually, type "python httpserver.py start"

Example One

We're testing account validation on a web app login page. Our first test will open a browser, go to the main page, and check the title. Then it will type in a correct username and password, click login, verify that the title is "Welcome Page", and that the page contains the text "Login succeeded". (Note: I'd start even more simply in real life – just open the browser for starters. But I have to try to keep this article a manageable length.)

The following is from my test, which is in a .txt file in real life (Robot Framework tests can be plain text or HTML). The asterisks on the section names must start in column 1. There are two spaces between the Selenium keywords such as 'input text', and the data or arguments used such as the field name 'username_field' and the value 'demo'. The "Test Cases" section includes all our test cases. The "Settings" section allows the test to access the correct libraries, and does tidying up.

*** Test Cases ***

Test Login

open browser <http://127.0.0.1:7272> firefox
title should be Login Page

input text username_field demo
input text password_field mode
Click Button login_button

Title should be Welcome Page
Page should contain Login succeeded

*** Settings ***

Library SeleniumLibrary
Test Teardown close browser

Example Two

Cool, we tested that a valid username and password can log in. But we have lots of test cases to cover. Valid and invalid combinations of usernames and passwords, empty ones, special characters, too-long and too-short ones, and more. Repeating the test above over and over with hard-coded values would get out of control pretty quickly. Plus, we want to test in multiple browsers and possibly test multiple servers.

When we see duplication, we always want to extract it out. Good test tools give you a way to do this. Robot Framework provides the concept of “keywords”. For example, we can use a keyword that navigates to the login page, a keyword to type in the username, and so on.

Here’s a first step towards extracting out duplication in our simple test. We have two test cases: “invalid account”, which tests an invalid username and password and verifies the error message, and “valid account”, which logs in with a valid username/password combination and verifies the landing page. We’ve put the values for browser and server into variables. We’ve defined keywords for actions such as typing in the username and password, clicking the submit button, and verifying the messages.

*** Test Cases ***

```
invalid account
  navigate to the login page
  type in username invalid
  type in password xxx
  click submit
  verify the invalid account message
```

```
valid account
  navigate to the login page
  type in username demo
  type in password mode
  click submit
  verify the valid account message
  click logout
```

*** Keywords ***

```
type in username [Arguments] ${username}
  input text username_field ${username}
```

```
type in password [Arguments] ${password}
  input text password_field ${password}
```

```
navigate to the login page
  log hello, world!
  open browser ${LOGIN PAGE} ${BROWSER}
  title should be Login Page
```

```
click submit
```

Click Button login_button

click logout

Click Link logout

verify the invalid account message

Title should be Error Page

Page should contain Invalid user name and/or password

verify the valid account message

Title should be Welcome Page

Page should contain Login succeeded

*** Settings ***

Library SeleniumLibrary

Test Teardown close browser

*** Variable ***

\${LOGIN PAGE} http://localhost:7272

\${BROWSER} firefox

This test is still sort of end-to-end-y, in that it first does the invalid case, then the valid one, then logs out. It's a step in the right direction. If the name of the password field in the HTML changes, we only have to change it in one place. Same thing if the text of the invalid login error message changes.

Example Three

It'd be nicer if we could just run the test with whatever username and password that we want to try, and check for the appropriate error message. One way to do this is by passing in variable values with command line arguments. We've refactored the test to expect variables to be passed in. We also separated out verifying the title and verifying the message,

*** Test Cases ***

login account

navigate to the login page

type in username \${username}

type in password \${password}

click submit

verify the title

verify the message

*** Keywords ***

navigate to the login page

open browser \${LOGIN PAGE} \${BROWSER}

title should be Login Page

\${title} = Get Title

should start with \${title} Login

```
type in username [Arguments] ${username}
  input text username_field ${username}
```

```
type in password [Arguments] ${password}
  input text password_field ${password}
```

```
click submit
  Click Button login_button
```

```
click logout
  Click Link logout
```

```
verify the title
  Title should be ${title}
```

```
verify the message
  Page should contain ${message}
```

```
*** Settings ***
Library SeleniumLibrary
Test Teardown    close browser
```

```
*** Variable ***
${LOGIN PAGE}    http://localhost:7272
${BROWSER}      firefox
```

We can supply the variable values from the command line:

```
./runDemo.py --variable username:demo --variable password:mode --variable
message:'Login succeeded' --variable title:'Welcome Page' ExampleThree.txt
```

I can easily visualize cranking a lot of username, password and message value combinations through this script. I could write a script to do this so I don't have to type them myself. My team uses a similar technique to test many combinations of data values with our Watir scripts. Different tool, same concept.

It turns out that in Robot Framework, running the test multiple times by passing in variables from the command line would be inefficient, because Selenium would start the browser up new each time. But this can be a good approach with other tools, and works well in Robot Framework for running the same test with different browsers (see http://robotframework-seleniumlibrary.googlecode.com/hg/demo/login_tests/invalid_login.txt for an example of this). Also, passing variables from the command line can be a powerful way to leverage automated tests to set up scenarios for manual exploratory testing.

Notice that I also split out the 'verify title' and 'verify message' test cases. That's a personal preference; I find that if my tests are more granular, it's easier to debug problems with the test itself. I didn't pass in the login page or browser values, but I could.

Example Four

There's still a lot going on in one test file. It would be nice if each of our keywords could be in its own little file. Good test tools allow this, and Robot Framework's implementation is called a 'resource'.

For example, we could take "Navigate to login page" and put it in a separate text file called "navigate_to_login.txt"

*** Keywords ***

```
navigate to the login page
  open browser ${LOGIN PAGE} ${BROWSER}
  title should be Login Page
  ${title} = Get Title
  should start with ${title} Login
```

Now we can simply refer to this file whenever we want to navigate to the login page in a test. In Robot Framework, we do this in the "Settings" section. Notice I also added some Documentation there.

*** Settings ***

Documentation Test account validation

Resource navigate_to_login.txt

*** Test Cases ***

```
login account
  navigate to the login page
  type in username ${username}
  type in password ${password}
  click submit
  verify the title
  verify the message
```

The Keywords section looks the same as before, except with the 'navigate to the login page' keyword removed.

If we're testing a web application, we're bound to have lots of tests that need to navigate to the login page. Now this functionality is encapsulated into one little file. Whenever something about navigating to the login page is changed, we only need to update one file, and all the tests that use it will still be able to do the navigation. We'd want to do something similar with the login account, so that our many GUI tests can simply include the login file in their Resource sections.

Design for Maintainability

I've only scratched the surface of test automation design principles. I didn't even get into patterns, such as "Build-Operate-Check", where we build our test data, operate on it, verify the results, and then clean up the data so the test can be rerun. But I hope I'm getting across my point: successful test automation requires good design skills, and they can be learned without too much pain.

I'm no expert at object-oriented design. My programming experience is in structured and interpreted languages, and with proprietary C-like test automation tool languages. The only OO scripting I've done is with Ruby, but I was able to succeed because my fellow tester was a Perl programmer and all my programmer teammates are willing to help too.

I've shown you some simplistic examples of how I personally would start extracting duplication out of a test script and start forming a library of basic test modules, like navigating to the login page and login. We could have modules for verifying HTML page titles, text, messages and elements, using variables for the names and values.

These concepts aren't easy to learn, but the investment pays off. My team and I have spent lots of time over the years refactoring the early tests I wrote in FitNesse and Canoo WebTest, because I made design mistakes or was ignorant of the tools' helpful features (and I had been automating tests reasonably successfully for at least a decade prior!) As a result of constantly learning, refactoring and improving, we're able to automate 100% of our regression testing without a heavy maintenance burden, and our tests have a high return on investment. When they fail, it's usually because there's a regression bug.

If you've read this far, you must be eager to learn more about test automation design. Ask a programmer on your team to pair with you to automate some tests, so you can learn some new techniques. Work through a good book such as *Everyday Scripting with Ruby* by Brian Marick. Search out blog posts and articles. Dale Emery has an excellent paper on Writing Maintainable Automated Acceptance Tests at http://dhemery.com/pdf/writing_maintainable_automated_acceptance_tests.pdf.

Your team can successfully automate regression tests. You'll need to invest some time and effort to experiment and learn the best approach. It's a journey, but an enjoyable one. Once you get on the right road, you'll have time for the other essential testing tasks you need to create a product your customers love.