HoodPopper

CS 1632 – DELIVERABLE 3:
Systems Testing a Web Application

Author:
Anthony Poerio (adp59@pitt.edu)

Github URL:
https://github.com/adpoe/HoodPopper

For our third deliverable in CS1632, our task was to write JUnit test code for the Selenium WebDriver. The code for this deliverable is designed as systems-level tests for web-application called HoodPopper.

As background, HoodPopper is a tool that a Ruby programmer can use to check exactly how ruby code is tokenized, parsed, and compiled. The user may enter valid ruby code into a text box on the HoodPopper homepage, and then she may ask the program to either: a) tokenize; b) parse; or c) compile, their input code. As such, HoodPopper is a way to "peek under the hood" of the Ruby interpreter, and thereby find potential errors in some given code, based on how compilation is performed.

The first issue faced with writing the systems-tests for the HoodPopper application was attempting to understand what end-users would be looking for exactly: What are their likely inputs, and what outputs would be relevant to test? In order to tackle this problem systematically, I decided to test each functionality separately. Thus, I divided my test suite into three sections (Tokenize, Parse, and Compile), and wrote tests specific to the expected functionality of each.

The next problem was finding a way to actually pass input to each of the three buttons, separately. The first button, "Tokenize", could be grabbed easily enough by simply running a findElement(by.name("commit")) command. But from there, I struggled with finding a way to target the other buttons, which also have the HTML tag "commit" in their names. I could only access Tokenize, which was the very first "*commit*" button on the page.

Eventually, I was able to target the other input buttons using **xpath**, though I know this method can sometimes be unreliable. Still, for me, it worked

and was relatively simple once I was able to understand how xpath regular expressions worked.

The final problem I encountered was finding a consistent way to check output on the results page, after submitting my code via any given button. I found that each button produced different outputs, so I first needed to mentally parse the outputs and find what I was looking for and wanted to test. And from there, I needed to find a way to reliably grab the information and parse it in my program itself.

After some trial and error, I discovered that running a findElement command on each result page, and search by the **tag name** "code" gave me the WebElement I was looking for. And then running **getText()** on that WebElement gave me a single String, containing all of the output info. From there, it was just a matter of using asserts with the Java contains() command to ensure that my desired output was present, and that undesired outputs were NOT present.

I was further able to refine my testing process by splitting the text segment I got back by either newlines or commas (depending on the functionality being tested), and then parsing each element/token/tree-node separately. This allowed me to run more complex edge and corner case tests, ensuring that **the exact number** of operators (for instance) that the user specified were correctly represented in the output.

Going forward, I would expect that testing more complex functionalities would be difficult primarily because it's sometimes hard to specify or find the *exact element* you are trying to test by ID, Tag, or other element that is simple to interface with in the HTML output. **More complex tests will require use of the**

xpath functionality, and that can be hard to understand and sometimes unreliable, as it is brittle to changes in the HTML output.

In the end, this process went smoothly, though getting the environment set up and getting oriented with the commands available via JUnit's Selenium interface was difficult at first.

To note: **None of my tests failed, but I did get many warnings from the Gargoyle Software CSS Error handler**. These warnings are represented in my confirmation outputs, seen on the next page of this document. But below these warnings, I got the confirmation that all tests passed, as expected.

To run the tests themselves, please make sure all of the Jar files I've uploaded are included in your target folder, and simply run the shell the shell script I've included, called "run.sh". From there, the tests will run, and you should be able to replicate my outputs.

Detail of All Tests Passing

Full Output, with Warnings. No errors.