Deliverable 3

For the third deliverable, we worked on beginning the implementation of our testing framework. This initial implementation and design would include the original five test cases that are listed below:

•testCase00.yml: "testCases/brokenParseFiles/ConflictingSpecError.yml"

•testCase01.yml: “testCases/brokenParseFiles/InvalidSpecTypeError.yml"

•testCase02.yml: "testCases/brokenParseFiles/InvalidSubSpecError.yml"

•testCase03.yml: "testCases/brokenParseFiles/MissingSpecError.yml"

•testCase04.yml: "testCases/brokenParseFiles/NullFieldError1.yml"

name: name of the test case
requirement: requirement being tested
component:
    class: class containing the method being tested
    provider: Python file providing the class above
method: method being tested
input:
    constructor: comma-separated arguments to the constructor
    method: comma-separated arguments to method
oracle: expected method return

The syntax of the test case is important, because it allows us to create reports in an HTML file to give the user useful information about how their code base fared based on the test case that they specified. The syntax of a test case file is verified to be correct before they are used to create a test case object. The user is notified if a test case is incorrectly written, so that they can return to the test case and re-write it using the proper syntax that we have provided.

We derive the path to the Python file in the provider field, and the class is directly imported using the class field. The true heart of the test case is in the constructor input field and method input field. These are the actual values that are used to instantiate objects from the python classes, and then to execute the methods.

After writing the test cases in the appropriate directory, TestAutomation/testCases, the user can then navigate to TestAutomation/scripts, and then run runSingularTest.sh or runAllTests.sh. The singular test script will take the path to a test case file as an argument, then report the results. The all tests script will take no arguments, and it will iterate through the test cases' directory, executing all test cases, and compiling the results into an HTML file for the user.

Detailed Instructions:

Set Up:

1. *git clone https://github.com/CSCI-362-01-2017/Fat-Mike-s-Pizza-Joint.git*
2. *cd Fat-Mikes-Pizza-Joint/TestAutomation/scripts/*
3. To allow testing of your own project's code, you will want to create a symlink in within the *TestAutomation/project/src/* directory.
    1. A symlink to this directory is provided for your convenience inside of *scripts/*.
    2. To create the symlink:
        1. *cd src/*
        2. *ln -s <path_to_your_projects_root>*
        3. *cd ..* (to return to *scripts/)*
4. Write your test cases in *TestAutomation/testCases* following the syntax. Again, for convenience, a symlink to this directory is provided inside of *scripts/*
5. You may want to create a symbolic link for your project's root directory in *TestAutomation/scripts* for your convenience

To Test Your Code Base:

1. Write test cases
2. navigate to *TestAutomation/scripts* directory
3. *./runAllTests.sh*
    1. This will run all test cases within the *testCases/* directory, and display the results as a report displayed in your web browser.
4. *./runSingularTest.sh testCases/testCaseX.yml*
    1. This will run a singular test case of your choosing and display the results as a report displayed in your web browser

To Set Up Fault Injection:

1. Have written test cases
2. Edit *TestAutomation/scripts/infected*
    1. This will contain a directory path, specifying where to look within your project files for modified code containing faults you have injected
3. *./createInfectedTests.sh*
    1. This will clone test case files into an infected test cases directory
4. *./runInfectedTests.sh*

1. this will run the tests form your infected test cases directory