TBD Sugar Labs Framework

Thomas Setzler, John-Tyler Cooper, Austin Purtell

Table of Contents

Introduction

This document will outline the purpose and usage of the testing framework designed by TBD. This framework is created to test methods from the Sugar Labs source code, specifically the functions.py code utilized by the Calculator application. The framework operates based on specialized drivers for each method. Data is accepted via specially formatted JSON files and then processed and output into an HTML file.

Within the output file, a timestamp notes the date and time in which the test was run. Each test case contains a test case ID, requirement description, inputs, oracle value, outputs, and whether the test case passed or failed. This HTML file is automatically opened into a browser for easy viewing.

Using the Framework

To utilize the framework, first ensure that a test case JSON is inserted into the test case file based on the format below.

```
{
        "id":"0000000",
        "test_name":"template",
        "drive_name":"driver name",
        "test_discription": "this is the template",
        "input":["2","2.0"],
        "oracle":"4.0",
        "outputs":[],
        "test_pass":false
    }
```

Once the test case is properly inserted, ensure a driver module for the method is inserted into the scripts/test_drivers folder and testDriver() within runAllTests.py is updated in order to account for the new driver.

If there is already an existing driver, simply upload the properly formatted test case JSON.

Once the test case JSON and driver are properly integrated into the framework, simply run the framework by running “python runAllTests.py” on the terminal within the TestAutomation/scripts directory.

Chapter 1

Sugar is a learning platform for children. It teaches kids how to interact and work a computer on their own. It has a bunch of interactive activities and puzzles. It's internet independent and runs on most computers running Linux, also can run on a Raspberry Pi. It has a simple UI and very easy to use. Sugar is developed with Python. Using Sugar Labs would be a good learning experience by being able to learn how to create simple games with Python.

Sugar Labs is the open source software which supports Sugar. Sugar Labs was worked on in the Google Summer of Code from 2016 to 2018. It is a member project of the Software Freedom Conservancy. According to Sugar Labs, Sugar is a collection of tools which they encourage the users to appropriate and create new activities to contribute to the network of Sugar users. It allows for children to write, read, or make music, classroom-learning style activities, automatic data backups in a “Sugar Journal,” and for easy open source activity sharing. Sugar aims to be friendly to teachers and children through a simple GUI and intuitive means of creating and experiencing content on their platform. They strive to create labs/learning environments around the world in regional ways by adapting Sugar activities to local curricula and language.

To begin working with Sugar Labs, we had to load an image of the live build into a virtual machine. We had some minor snags in the installation, which made us optimistic about the tasks to come. Surprisingly, learning to navigate our way sugar labs proved to be more difficult than expected. Sugar Labs has its own terminal as well as a bunch of development tools. We found the built-in test cases, but we were not sure that we were able to run them properly. We got in touch with a GitHub contributor, asking him about running testing. He was under the impression that we intended to contribute to the source, and also suggested that we did not have the resources available to learn the required things needed for running the Sugar Toolkit unit tests. He also gave us very relevant and important information on how many lines of code each of the Sugar components contained.

However, it did allow us to infer how we would be interacting with this software, because we are going to need to choose a component to run testing on. The options are Sugar Toolkit (Developer Tools), Sugar (Actual software), Datastore (the automatic backups), Core Sugar Activities (pre-made activities that run on Sugar), and the library of public contributors to Sugar (a lot of lines of code). We could probably run a series of unit tests on one of the Sugar activities alone, however, whether this would fit within the scope of the project is up for discussion. The GitHub contributor we got into contact with suggested looking into the Datastore component, probably because it contains the least amount of lines. The tests that we found in the installation are for Sugar itself, and supposedly should be able to be run by anyone who uses Sugar for testing purposes. Although, it seems rather strange that they would

not contain any logging or output when the test case runs properly, as we found through our experimentation.

Eventually, we were able to run the tests and identify the testing procedure in Sugar Labs. Sugar Labs has several pre-existing test cases that are prebuilt into the installation. These test cases are designed to test major functions of the Sugar Labs software. Upon a successful test, the test case exits normally and often nothing is output to the terminal. Some test cases, such as test_mime.py, which output the success or failure of a test along with the time it took to complete. Accessing the test cases is not difficult as each module's prebuilt tests are stored within the /tests/ directory within said module's directory. Within the test directories, occasionally subdirectories will exist with supplemental files needed for testing. One test, test_backup.py, tests where the backup exists and is functioning properly. Another test, test_user_profile attempts to create multiple, varied user profiles. If all tests provide acceptable results, then the user can imply that Sugar Labs is operating normally. After running that testing, we attempted to get Sugarizer running. Sugarizer is an application-based GUI that runs Sugar. We got Sugarizer running on Google Cloud Platform and ran some tests. We launched a GCP cloud instance, started up a web server, and ran Sugarizer from there as a web application. We then ran the test/index.html and tested the mochajs unit test on the datastore functions. This was a much easier environment for us to test in since it uses nodejs. It also allows us to see passes and failures in more detail. Ideally, this is the Sugar environment that we would be able to conduct our own testing in.

Chapter 2

Once our environment was established, we focused on identifying our methods for testing and determining how we would go about doing it. We started by searching through the various GitHub pages which make up the Sugar project. We decided on the main SugarLabs GitHub, where we found an integrated activity called calculate-activity. This activity uses functions.py, which is a collection of calculator methods for doing math problems (all methods available in calculate activity). It includes a means for converting values to decimal objects, as well as several different kinds of mathematical operations. We decided that this would be a manageable set of functions for testing.

We then looked through the series of functions in functions.py and identified which candidate we would begin with. We decided on the divide method because we figured it would be an easy operation to identify test cases for and observe the results, potential errors included. We then had to determine which values we could pass into the method to achieve a well-balanced variety of test cases. Although it is difficult to cover every potential input into five methods, we decided to use positive over negative integers, negative over negative integers, division by zero, overflow value, and invalid type input. The first two are general tests and the last three focus on what the function returns in the case of a problematic input. We also plan to ensure that our test cases satisfy the if statements within the function. We then organized the template for storing our input parameters and output. We structured it using JSON format. It contains the id, test name, test description, inputs, expected outputs, outputs, and whether the test passed or not. For example: { "id":0000001, "test_name":"Test Case Template", "test_discription": "This is the template", "inputs":{ "input1":[0,1], "input2":[1,1], "input3":[2,1], "input4":[0,0], "input5":[1,-1] }, "oracles":{ "oracle1":0, "oracle2":1, "oracle3":2, "oracle4":0,

"oracle5":-1 } "outputs":[], "test_pass":[false,false,false,false,false] } We may also log the time at which the output is calculated.

Chapter 3

The first task we had to focus on was reworking our test case templates. We had the templates set up so that we were testing each test case for a method within a single template. We now understand that each template must correspond to a single test case. Therefore, we should have twenty-five test case text files by the end of the project. For this deliverable we have altered our files to now have five test case files. Here is an example test case file: {

"id":"0000001",

"test_name":"div(x, y)",

"drive_name":"",

"test_discription": "Testing div(x, y) in the functions.py",

"input":["5","-1"],

"oracle":"-5",

"outputs":[],

"test_pass":false}

We have also constructed a driver in which we will be running these tests. We originally thought that there would only be one driver for each method being tested. We now understand that we will have to have separate to test each method in functions.py.

We altered our design so that the original test case files are untouched. When a test is executed, it will output the results into the result directory in the form of an HTML page and then open the page.

In our _runDivTest(testCase) method, we realized that we were having an issue with type casting using our JSON files and the output from the div() method in functions.py. We made all of the values in the JSON file strings, and we cast the inputs to integers or floats, as necessary, after reading them in the driver, contingent on whether they have a period in them. The oracle from the test case file remains a string, and the output from the div method is type casted as a string for comparison with the oracle.

After restructuring our directory architecture, we had troubles with importing modules into the drivers. The modules were not being recognized by the drivers so we decided to undo our architectural changes until we could fix the issue.

Chapter 4

Our next task began by restructuring our file architecture to remove the intermediate TestScripts repository in TestAutomation/Scripts/ that housed all the code for running the framework. By removing this intermediate repository and shifting the contents of said repository up a level, we were able to remove the intermediate TestSugarFunctions.py script that was being used to operate the testing framework.

Once our file structure was improved, we chose four more methods from within Sugar Lab's functions.py and created drivers and test cases for each of the methods. The chosen methods are factorize(), factorial(), mod(), and pow(). These methods were chosen for testing to ensure that they account for varying forms of input while still maintaining their accuracy and functionality.

During the framework expansion, a few changes were made to the overall design of the framework. The template was restructured somewhat to remove unnecessary whitespace that caused issues when parsing. The HTML display was changed to remove the description category for each test case because the sections was redundant and unnecessary.

Chapter 5

For this task, we were instructed to inject 5 faults into the code that we are testing, functions.py. In the div() method, we commented out the first if statement that checks if the denominator is zero. Within the factorial() method, we commented out the first if statement which checks if the number is less than zero. Also within the factorial() method, we inverted the if statement which checks if n is zero to check if it is not zero. For the mod method, we inverted the if statement which checks if y is an integer, so it now checks if y is not an integer. In the pow() method, we changed the type-casting in the return method to cast to integers, rather than floats.

This resulted in the failure of 11 test cases:

1 in div

2 in pow

4 in factorial

4 in mod (1 in mod already fails)

We assumed commenting out the if statements would be the most direct means of injecting a fault into the code, so we did that where applicable. However, it is clear that in the case of the mod() method, all of the tests that would normally pass would fail, since we inverted the only two return paths in the method. We were also having some issues injecting faults into the natural log method, so we decided to not alter it.

During this deliverable, we also made some improvements and recommended revisions to our code. The results HTML was updated to display the requirements description for each test case. We also changed the oracle of the last test case in mod so that it now fails, as that is the expected result. Sugar Lab's code throws the error "Can only calculate x modulo integer" when x is a string. There were issues with type-casting within the individual drivers, so in order to improve efficiency and fix this problem, we handle all initial type casting within the master driver, before inputs are passed into the individual drivers. Each driver was refactored to reduce code redundancy by preloading all JSON data in the master driver, then passing inputs and oracles values to the drivers. After the drivers receive the necessary inputs and oracle, they run the method based on according inputs, and compare the outputted data to the oracle. The methods returned whether the test case passed or failed, and the computed output. Finally, we changed the main driver to run each test driver individually and not load them all at once into a list.

Chapter 6

This project has enhanced our knowledge by providing valuable lessons. The challenges encountered throughout the project while frustrating, helped to develop our skills. We have learned a lot about software engineering and teamwork.

Within software engineering, we learned how to use relative pathing for importing files and modules for usage. We learned how to read and utilize data from JSONs. Modularization played a key role in the creation of the framework and we had to learn how to properly utilize the modules once created. These modules allowed for both increased efficiency and minimal code reuse.

For teamwork, we realized that by assigning roles based on everyone's unique skill sets, they can lead and teach the other members. Thomas was designated the project lead to keep the project organized, on schedule, and efficient methods to meet the requirements. John-Tyler was designated the engineer to oversee the automation efforts to maximize efficiency within the code. Austin was designated the scribe to oversee front-end displays and oversee official communications. Each group member participated in all areas but utilized the knowledge of whoever was most familiar with the current problem. Slack was utilized for efficient communication when no one was able to meet at the same time.

Self-Evaluation

Our work was challenging and convoluted at times. In the beginning we were not creating efficient code, we were focused on code that works and is easily usable within a virtual machine. After the first two deliverables, we focused on making higher quality code so that we would no longer have to fix numerous issues every time new features were added. Our organization improved over the course of the semester to reflect both the required directory organization scheme and remove unnecessary and redundant directories. As the semester progressed, our skills progressed too, which allowed once difficult tasks to be completed both easily and efficiently. At times we struggled to meet due to everyone's vastly differing schedules, but we were able to overcome that by making appropriate use of both class time and Slack channels. Over all all three team members contributed their fair share of effort and communication between us was good.

Project Evaluation

Overall this project was well structured and very useful. By initially requiring us to complete tasks with minimal preparation, we were forced to work together to overcome unknown difficulties. The pace of the deliverables was well spaced so that as the semester went on and skills developed, the time between deliverables matched to reflect that instead of remaining constant, which encouraged us to continue to improve. However, the time between deliverables 4 and 5 could be slightly longer and the time between 2 and 3 could be shortened slightly. Overall the assignment structure made sense, starting out slow but difficult and gradually building upon each previous assignment to create a functioning testing framework.