Sugar Labs: Calculate Activity Automated Testing Framework

Fantastic Four:
Patrick McCabe
David Thompson

Fall 2020 Team Term Project
CSCI-362-01
Dr. Jim Bowring

Contents

Introduction

Over the course of the Fall 2020 semester, we worked on a team term project to design and implement a testing framework for an open source software project. At the start of the semester, we were given the opportunity to choose which project we would work on from a selection of HFOSS projects. Our priorities for selecting a project were based on programming languages, preferring a project in Python over one in Java over one in another language. We ultimately settled on Sugar Labs, a Python project that is focused on helping children learn through software. Once we had selected which project to work with, we proceeded with building our testing framework in a series of scheduled deliverables, presented in class, which we tracked on our team wiki in our team Github repository. By semester's end, we had a functioning framework with twenty-five test cases covering five different methods. The framework is run with a main scripts that calls on a series of helper scripts to read a test case, test it with a corresponding test driver, and produce an html report of the collected results.

Chapter 1: Building Sugar Labs

Our first task was to clone the repository of our chosen HFOSS project and build it. We produced the following deliverable, presented on September 22:

Summary:

After testing the sugar project across multiple virtual machines, we were unable to successfully build the sugar project. We were successful with an install of sugar on an Ubuntu 19.04 Virtual Machine by installing the `sucrose` package. We will continue to attempt to build the software from source.

Technical Description:

We have created Virtual Machines for the following versions of **Ubuntu**:

- 20.04 `Focal Fossa`
- 19.04 `Disco Dingo`
- 18.04 `Bionic Beaver`
- 17.04 `Artful Aardvark`

Once we have set up the VMs, we have cloned our repository to the VM and run the following scripts:

clone.sh
```
#!/bin/bash

for module in sugar{-datastore,-artwork,-toolkit,-toolkit-gtk3,}; do
    git clone https://github.com/sugarlabs/$module.git
done
```

deps.sh
```
#!/bin/bash

for module in sugar{-datastore,-artwork,-toolkit,-toolkit-gtk3,}; do
    sudo apt build-dep $module
done
sudo apt install python{,3}-six python3-empy
```

```
sudo apt install autoconf autogen intltool libtool automake autotools-dev libopts25
libopts25-dev \
libglib2.0-dev gtk+-3.0 librsvg2-dev libasound2-dev python-empy GTK+-2.0 python2.7-
dev python-dev \
gtk2.0 icon-naming-utils

build.sh
#!/bin/bash

for module in sugar{-artwork,-toolkit-gtk3,-datastore,}; do
    cd $module
    ./autogen.sh --with-python3
    make
    sudo make install
    cd ..
done
```

We have run into issues with our `deps.sh` script which installs all of the dependencies for the `sugar` repository to build. Alongside trying to build the project from source, we installed the `sucrose` package in the Ubuntu Aptitude (apt) repositories and as per the instructions on the Github Repository, logged out and the desktop environment was only available in the Ubuntu 19.04 Virtual Machine once the package was installed via the repository.

Chapter 2: Detailed Test Plan

Our next task was to produce a detailed test plan for our project. The plan would include five of the eventual twenty-five test cases for our framework. We produced the following deliverable, presented on October 6:

Updates since Deliverable 1:

Since **Deliverable 1**, we have scrapped the plan of building the entire desktop environment for `sugar` and instead have shifted our efforts to testing individual `Sugar Activities`. The first activity we have selected to test is the `calculate-activity`.

Summary:

We are drafting an initial test plan for individual `Activities` from `sugar` due to dependency issues with required packages. Our initial tests for the `calculate-activity` are focused on input validation and making sure that there are no errors that arise with specific inputs into the `add()` function of the python class.

Test Plan:

Process:

The tests for this software project will utilize the `bash` command line interface and using the `python` interpreter that `sugar` uses to run the `calculate-activity`. The `calculate-activity` consists of many methods that are used to compute results that the user would like to compute.

Requirements Traceability:

The requirements of a calculator with all its basic functions, for these cases specifically the addition function, further more clarify that we want to be able to do addition using all real numbers with the test cases testing natural numbers, whole numbers, integers, rational, and real numbers.

Tested Items:

The following functions are set to be tested with the possibility of more functions being added at a later date:

1. add(x, y)
2. sub(x, y)
3. mul(x, y)
4. div(x, y)
5. pow(x)

Testing Schedule:

The following is our testing schedule:

Date	Deliverable Number	Task
Nov. 5	3	Design an automated testing framework
Nov. 17	4	Create 25 test cases and run them with our framework
Nov. 24	5	Create repeatable faults in the application and identify tests that are planned to fail

Hardware and Software Requirements:

Hardware: physical or virtual machine with appropriate resources that can run a Linux Distribution

Software: a functional terminal emulator with the `bash` environment, `python` interpreter, and the ability to install dependencies if needed through a package manager

Constraints:

The classes and methods must not rely upon the `python-sugar3` package or library, or any other broken dependency or package

Systems Tests:

Please see the chart below with the initial test cases for the `calculate-activity`

Test Cases:

Test ID	Requirement	Component	Method	Test Inputs	Expected Outcomes	Text file
1	Addition of Two Numbers	`functions.py`	add(x, y)	(1,2)	No errors expected with natural numbers, result should be returned as **3**	Test Case 1
2	Addition of Two Numbers	`functions.py`	add(x, y)	(-1,1)	No errors expected with integers, result should be returned as **0**	Test Case 2
3	Addition of Two Numbers	`functions.py`	add(x, y)	(1.1,2.1)	No errors expected with rational numbers due to casting with the `_d` function, result should be returned as **3.2**	Test Case 3
4	Addition of Two Numbers	`functions.py`	add(x, y)	(pi,sqrt(2))	No errors should occur with irrational numbers, the result should be returned as `math.pi + math.sqrt(2)`	Test Case 4
5	Addition of Two Numbers	`functions.py`	add(x, y)	(sys.maxsize,2)	**Note: the included `sys` library is needed to get the maximum integer** No errors occur when invoking the maximum size integer added with 2, the result will vary based on the system architecture and will add 2 with no overflow error	Test Case 5

Chapter 3: Initial Test Automation

Our third task was to design and begin building our testing framework, implementing at least five of the eventual twenty-five test cases. We produced the following deliverable, presented on November 5:

Executive Summary:

Our automated testing framework of the `sugarlabs calculate-activity` is 25% complete with five test cases developed to test the `add(x, y)` function within the activity. The test cases follow the table below:

Test Cases:

Test ID	Requirement	Component	Method	Test Inputs	Expected Outcomes	Text file
1	Addition of Two Numbers	`functions.py`	add(x, y)	(1,2)	No errors expected with natural numbers, result should be returned as **3**	Test Case 1
2	Addition of Two Numbers	`functions.py`	add(x, y)	(-1,1)	No errors expected with integers, result should be returned as **0**	Test Case 2
3	Addition of Two Numbers	`functions.py`	add(x, y)	(1.1,2.1)	No errors expected with rational numbers due to casting with the `_d` function, result should be returned as **3.2**	Test Case 3
4	Addition of Two Numbers	`functions.py`	add(x, y)	(pi,sqrt(2))	No errors should occur with irrational numbers, the result should be returned as `math.pi + math.sqrt(2)`	Test Case 4
5	Addition of Two Numbers	`functions.py`	add(x, y)	(sys.maxsize,2)	**Note: the included `sys` library is needed to get the maximum integer** No errors occur when invoking the maximum size integer added with 2, the result will vary based on the system architecture and will add 2 with no overflow error	Test Case 5

We will look to add in the following methods:

- `sub(x, y)`
- `mul(x, y)`
- `div(x, y)`
- Additional method TBD

Technical Summary:

Directory Descriptions:

1. `docs`: Home to the documentation for the project as well as a `README` for running the project
2. `oracles`: Home to the outputs of our test cases
3. `project`: Home to the `sugar-activity` code that we will be testing
4. `reports`: Home to all the reports generated by our test cases
5. `testCases`: Home of all the test case files with the following format:

- [Test Suite ID]
- [Test Case ID]
- [Requirement]
- [Driver]
- [Component]
- [Method being tested]
- [Inputs (comma separated)]
- [Expected Oracle]

6. `testCaseExecutables`: Home to the drivers for the methods being tested for low coupling and high cohesion in the testing framework.

Instructions on how to run our Testing Framework

Prerequisites:

1. A Linux based system with the `bash` command line interface

2. Python3 interpreter
3. The `git` program is installed on the system

Setting up the environment:

1. On the linux system, open a terminal emulator
2. Clone our repository with the following command: `git clone https://github.com/csci-362-01-2020/Fantastic-Four.git`
3. Navigate to the propper directory: `cd Fantastic-Four/TestAutomation`
4. Run the test driver: `./scripts/runAllTests.py`

Chapter 4: Completed Testing Framework

Our fourth task was to complete the implementation of our testing framework and have all twenty-five required test cases automatically tested by our script. We produced the following deliverable, presented on November 17:

Executive Summary

We have completed the task of creating 20 additional test cases in addition to the test cases from the last deliverable. The new test cases are the following `testCase6.txt-testCase25.txt`. All the test cases test the following methods:

1. `add(x, y)`: addition of two numbers
2. `sub(x, y)`: subtraction of two numbers
3. `mul(x, y)`: multiplication of two numbers
4. `div(x, y)`: division of two numbers
5. `pow(x, y)`: the exponentiation of two numbers

Test Cases:

Test ID	Requirement	Component	Method	Test Inputs	Expected Outcomes	Text file
1	Addition of Two Numbers	`functions.py`	add(x, y)	(1,2)	No errors expected with natural numbers, result should be returned as **3**	Test Case 1
2	Addition of Two Numbers	`functions.py`	add(x, y)	(-1,1)	No errors expected with integers, result should be returned as **0**	Test Case 2
3	Addition of Two Numbers	`functions.py`	add(x, y)	(1.1,2.1)	No errors expected with rational numbers due to casting with the `_d` function, result should be returned as **3.2**	Test Case 3
4	Addition of Two Numbers	`functions.py`	add(x, y)	(pi,sqrt(2))	No errors should occur with irrational numbers, the result should be returned as `math.pi + math.sqrt(2)`	Test Case 4
5	Addition of Two Numbers	`functions.py`	add(x, y)	(sys.maxsize,2)	**Note: the included `sys` library is needed to get the maximum integer** No errors occur when invoking the maximum size integer added with 2, the result will vary based on the system architecture and will add 2 with no overflow error	Test Case 5
6	Subtraction of Two Numbers	`functions.py`	sub(x, y)	(1,2)	No errors expected with natural numbers, result should be returned as **-1**	Test Case 6
7	Subtraction of Two Numbers	`functions.py`	sub(x, y)	(-1,1)	No errors expected with integers, result should be returned as **-2**	Test Case 7
8	Addition of Two Numbers	`functions.py`	sub(x, y)	(1.1,2.1)	No errors expected with rational numbers due to casting with the `_d` function, result should be returned as **-1**	Test Case 8
9	Subtraction of Two Numbers	`functions.py`	sub(x, y)	(pi,sqrt(2))	No errors should occur with irrational numbers, the result should be returned as `math.pi - math.sqrt(2)`	Test Case 9
10	Subtraction of Two Numbers	`functions.py`	sub(x, y)	(sys.maxsize,2 * (sys.maxsize + 1))	**Note: the included `sys` library is needed to get the maximum integer** No errors occur when invoking the maximum size integer subtracted by 2, the result will vary based on the system architecture and will subtract 2 with no overflow error	Test Case 10

11	Multiplication of Two Numbers	`functions.py`	mul(x, y)	(1,2)	No errors expected with natural numbers, result should be returned as **2**	Test Case 11
12	Multiplication of Two Numbers	`functions.py`	mul(x, y)	(-1,1)	No errors expected with integers, result should be returned as **-1**	Test Case 12
13	Multiplication of Two Numbers	`functions.py`	mul(x, y)	(1.1,2.1)	No errors expected with rational numbers due to casting with the `_d` function, result should be returned as **2.3100000000000005**	Test Case 13
14	Multiplication of Two Numbers	`functions.py`	mul(x, y)	(pi,sqrt(2))	No errors should occur with irrational numbers, the result should be returned as `math.pi * math.sqrt(2)`	Test Case 14
15	Multiplication of Two Numbers	`functions.py`	mul(x, y)	(sys.maxsize,2)	**Note: the included `sys` library is needed to get the maximum integer** No errors occur when invoking the maximum size integer multiplied with 2, the result will vary based on the system architecture and will multiply 2 with no overflow error	Test Case 15

16	Multiplication of Two Numbers	`functions.py`	div(x, y)	(1,2)	No errors expected with natural numbers, result should be returned as **0.5**	Test Case 16
17	Division of Two Numbers	`functions.py`	div(x, y)	(-1,1)	No errors expected with integers, result should be returned as **-1.0**	Test Case 17
18	Division of Two Numbers	`functions.py`	div(x, y)	(1.1,2.1)	No errors expected with rational numbers due to casting with the `_d` function, result should be returned as **0.5238095238095238**	Test Case 18
19	Division of Two Numbers	`functions.py`	div(x, y)	(pi,sqrt(2))	No errors should occur with irrational numbers, the result should be returned as `math.pi / math.sqrt(2)`	Test Case 14
20	Division of Two Numbers	`functions.py`	div(x, y)	(sys.maxsize,2)	**Note: the included `sys` library is needed to get the maximum integer** No errors occur when invoking the maximum size integer divided by 2, the result will vary based on the system architecture and will divide 2 with no overflow error	Test Case 20

21	Exponentiation of Two Numbers	`functions.py`	pow(x, y)	(1,2)	No errors expected with natural numbers, result should be returned as **2**	Test Case 21
22	Exponentiation of Two Numbers	`functions.py`	pow(x, y)	(-1,1)	No errors expected with integers, result should be returned as **-1**	Test Case 22
23	Exponentiation of Two Numbers	`functions.py`	pow(x, y)	(1.1,2.1)	No errors expected with rational numbers due to casting with the `_d` function, result should be returned as **1.2215876651600335**	Test Case 23
24	Exponentiation of Two Numbers	`functions.py`	pow(x, y)	(pi,sqrt(2))	No errors should occur with irrational numbers, the result should be returned as `math.pi ** math.sqrt(2)`	Test Case 24
25	Exponentiation of Two Numbers	`functions.py`	pow(x, y)	(sys.maxsize,2)	**Note: the included `sys` library is needed to get the maximum integer** No errors occur when invoking the maximum size integer raised with 2, the result will vary based on the system architecture and will raise sys.maxsize to the power of 2 with no overflow error	Test Case 25

Failed Test Cases:

Test Case ID	Expected Oracle	Actual Oracle	Reason for Failure
16	0.5	1/2	Different output generated by the `div(x, y)` method
17	-1.0	-1	Precision is different for `div(x, y) method`
23	1.2215876651600335	1.221587665160033476	Rounding for the `pow(x, y)` method
24	5.047497267370911	5.0474972673709110 89	Rounding for the `pow(x, y)` method

Chapter 5: Fault Injection

Our final task was to design and inject five faults into our source code. The faults were to cause at least five test cases to fail, but not cause all test cases to fail. We produced the following deliverable, presented on November 24:

Executive Summary:

We have completed the task of injecting faults into our source code. A one-symbol change to emulate a typo was made in the return statement for each of the following methods:

1. `add(x, y)`: "+" was replaced with "-", to simulate an adjacent key typo
2. `sub(x, y)`: "-" was replaced with "+", to simulate an adjacent key typo
3. `mul(x, y)`: an additional "*" was added, to simulate a double keystroke typo
4. `div(x, y)`: an additional "/" was added, to simulate a double keystroke typo
5. `pow(x, y)`: a "*" was removed, to simulate a missed keystroke typo

Small tweaks were made to a few test cases to ensure that not all tests will fail when the faults are injected.

Technical Summary:

Fault Injection Process

Faults were injected locally but not pushed to the repository so that a fresh clone of the repo will have clean source code. Directions for duplicating our faults can be found in the `faults.txt` file, which can be found in the directory `Fantastic-Four/TestAutomation/docs`.
An example report generated by our framework while faults are present, `faultReport.html`, can be found in the directory `Fantastic-Four/TestAutomation/reports`.

Fault Injection Results

Expected Results:

All previously passing test cases now fail, with the following exceptions, which are unaffected:

Test ID	Requirement	Component	Method	Test Inputs	Oracles
11	Multiplication of two whole numbers	`functions.py`	mul(x, y)	(2,1)	2
12	Multiplication of negative and positive integer	`functions.py`	mul(x, y)	(-1,1)	-1
21	Exponentiation of two whole numbers	`functions.py`	pow(x, y)	(2,1)	2
22	Exponentiation of negative and positive integer	`functions.py`	pow(x, y)	(-1,1)	-1

Unexpected Results:

The following test cases are failing, with or without fault injection, but should pass with or without faults:

Test ID	Requirement	Component	Method	Test Inputs	Oracles
16	Division of two whole numbers	`functions.py`	div(x, y)	(2,1)	2
17	Division of negative and positive integer	`functions.py`	div(x, y)	(-1,1)	-1.0

Chapter 6: Final Thoughts

Experiences and Lessons Learned

Over the course of the semester, we were able to learn some new technical skills and improve upon existing ones. To complete our project, we needed to work with Git / Github, Linux, and Python3. We worked in a command line terminal and wrote a number of scripts. Since not all team members had equal experience in these areas, we also got to experience working to get on the page at a technical level. For example, it was much easier to just jump in and work with the Python code, since Python was not new to anyone. Terminal commands, on the other hand, required more teamwork.

We also got a first-hand look at some of the common software engineering setbacks that we discussed throughout the course. We experienced technical setbacks (the inability to work with much of the Sugar Labs environment due to the missing `python-sugar3` package) as well as personnel setbacks (at project start, our team had four members; by the end, we had two).

Project Evaluation

Overall, we were pleased with our final product; we created a functioning framework that runs from the command line and uses a series of scripts to evaluate provided test cases and generate a testing report. However, we did identify some aspects of our design that we would update if we did the project over again. The way our scripts call our drivers is one area that needs improving; the time and memory efficiency of generating a report would also need to be improved upon to accommodate a very large number (thousands to millions) of test cases. Another design flaw comes with our testing intentions: we designed the framework with the intention of testing mathematical functions for correct number outputs. Our framework meets this goal quite

successfully, however, if we wanted to add test cases for a wider variety of potential issues, we would need to update the portion of our script that evaluates whether a test passed or failed.

Assignment Evaluation

The term project assignment was mostly straightforward and easy to follow. There were a few instances, such as deliverable and final report descriptions, where more detailed specifications would have made it easier to make sure we were completely fulfilling assignment expectations. There were also some feedback inconsistencies; we had a few aspects of our framework that we presented and were told were up to standard, only to be told a deliverable or two later that those same aspects were not actually meeting expectations. We observed this feedback issue with other teams as well. Again, more detailed project specifications might help ease this issue.