

# **Automated Testing Framework**

## **Sugar Labs**

Team-3

Brielen Beamon: [beamonbh@g.cofc.edu](mailto:beamonbh@g.cofc.edu)

Logan Smith: [smithla1@g.cofc.edu](mailto:smithla1@g.cofc.edu)

Scott White: [whites2@g.cofc.edu](mailto:whites2@g.cofc.edu)

Dr. Bowring

CSCI 362

December 1, 2016

# Table of Contents

Title Page	1
Table of Contents	2
<b>Introduction</b>	<b>3</b>
<b>Chapter 1</b>	<b>4</b>
Choosing a Project	4
Building and Running Sugar Labs	4
Organizing Our Project	5
<b>Chapter 2</b>	<b>6</b>
Requirements Traceability	6
The Testing Process	6
Tested Items	6
Test Case Format	6
Test Recording Procedure	7
Hardware and Software Requirements	8
Constraints	8
Testing Schedule	8
<b>Chapter 3</b>	<b>9</b>
Automated Testing Framework	9
Automated Testing Framework Architecture	9
Using The Automated Testing Framework	10
<b>Chapter 4</b>	<b>12</b>
Our Test Cases	12
<b>Chapter 5</b>	<b>19</b>
Fault Injection	19
<b>Chapter 6</b>	<b>21</b>
Our Experiences	21
What We Learned	21
Self-Evaluation	21
<b>Glossary</b>	<b>22</b>

# Introduction

Our project is centered around [Sugar Labs](#), a free and open source software project that focuses on allowing people around the world to learn more about the world around them. Sugar Labs is a spin off of [One Laptop Per Child](#), constituting a simple Linux-based operating system. It is used to give disadvantaged populations an opportunity to interact with a simple computer system.

To test Sugar Labs, we designed an automated testing framework that will run a series of test cases and produce a report detailing the results of these test cases. The test cases we have written are intended to verify that the source code is error free and fully functioning. As mentioned above, Sugar Labs is a complete operating system which complicates the process of testing the system as a whole. To this end, we decided to verify the correctness of a critical activity in Sugar Labs; the calculator activity.

This was accomplished by picking some key methods in the calculator activity and ensuring these methods gave the correct output for whatever input we provide. In this, we attempted to utilize wide testing partitions to ensure as much test coverage as possible. Once we specified these test cases, we used our automated testing framework to execute these tests and compile the results so we might analyze them. Through our testing, we are confidently able to conclude that the calculator activity of Sugar Labs works fully and correctly.

# Chapter 1

## Choosing A Project

We debated between a number of projects, notably Sugar Labs, Martus, and Amara. We settled on using Sugar Labs for a couple of reasons. We wanted to work with a project that was written in Python, and we also liked the cause that Sugar Labs served. By allowing disadvantaged populations access to tools to learn and explore the vast wealth of knowledge humanity has acquired, we believe that this software can have an incredible impact. Thus, we wanted to help insure it works for those who use it.

## Building And Running Sugar Labs

From the Sugar Labs Github page, we forked a clone of their code. Once we had done so, we began the processing of building and running the software. We ran into some issues at this point. There is a bug in the current release version of the software, which causes the system to crash when trying to build it. As a temporary fix, we edited a JSON configuration file used to determine which components should be loaded in the build process. Our edits caused the software to not use the faulty component, and once we made this edit, Sugar Labs was able to run – albeit with reduced functionality – and we began the process of trying to run their test cases. It was here that we ran into a great deal of trouble. Most of the Sugar Labs test cases required large numbers of dependencies that we were unable to set up. It is our belief that since Sugar Labs is meant to run as an operating system, rather than a stand-alone program, the dependencies required by the various activities are organized in such a way that makes them difficult to utilize when attempting to run individual components of the software project. In response, we focused on the activities that had the fewest numbers of dependencies.

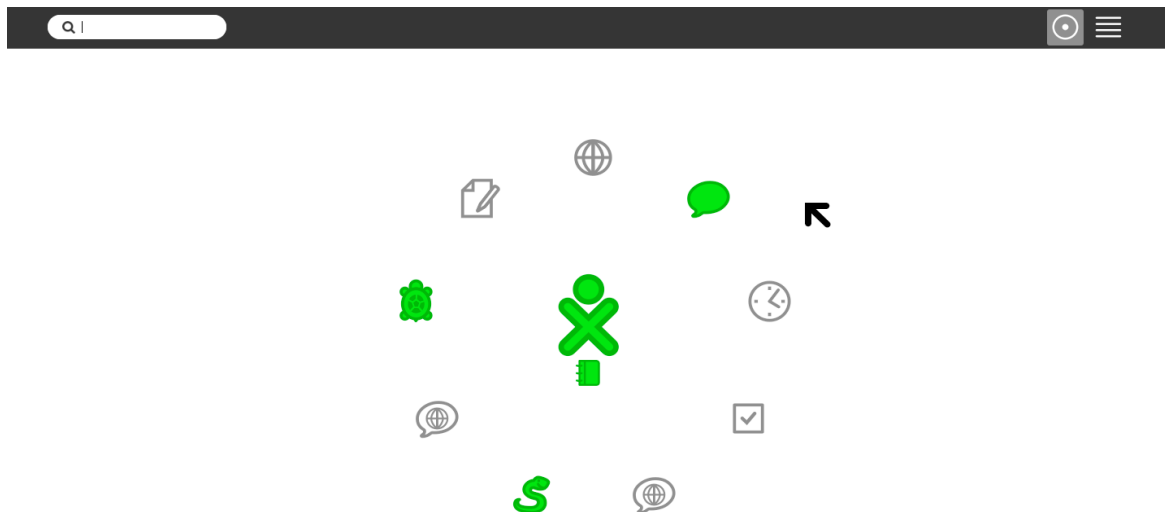


Figure 1: The home screen of the Sugar Labs open source software project

## Organizing Our Project

As per the directives given for this assignment, we will be using a very specific directory structure to store and organize our work. The format of our directory structure will be as follows:

```
/TestAutomation
  /project
    /src
    /bin
    /...

  /scripts
    runAllTests.py
    ... other helper scripts

  /testCases
    testCase1.txt
    testCase2.txt
    ...

  /testCasesExecutables
    testCase1(may be folder or file)
    ...

  /temp (for output from running tests)
    testCase1results (might be folder or file)
    ...

  /oracles
    testCase1Oracle (might be folder or file) ...

  /docs
    README.txt

  /reports
    testReport.html
```

## Chapter 2

### Requirements Traceability

As Sugar Labs is intended for people without access to other methods of learning, it is imperative that the information provided by Sugar Labs is accurate. For a person engaged in learning, nothing can be more damaging than viewing and committing to memory false information and knowledge. In addition, those using Sugar Labs may have no other way to verify what they learn through the activities therein. It is with that in mind that we come to the most important requirement (that we will trace our test cases too) for Sugar Labs: accuracy/reliability.

### The Testing Process

Our testing shall focus on the calculator activity of Sugar-Labs. We plan to observe the behavior of that particular activity and verify that it functions as it should. In order to do this, we plan to write a variety of test cases that establish correct behavior on both normal and edge cases.

### Tested Items

We will be testing the calculator activity. More specifically, we will be testing the file `functions.py`, which handles all the calculations for the calculator. At this point, the individual methods we are testing include:

- `pow(x, y)` — returns  $x$  raised to the  $y$  power
- `add(x, y)` — returns  $x + y$
- `div(x, y)` — returns  $x / y$
- `sqrt(x)` — returns  $\sqrt{x}$
- `cos(x)` — returns the cosine value of  $x$

### Test Case Format

The test cases are plain text files (.txt) that contain all the information the automated testing framework needs to test a segment of the Sugar Labs source code. These test cases need to follow a strict format to be recognized and used by the automated testing framework, but as long as they do, any number of test cases can be processed by the automated testing framework. The format of the test cases is as follows:

#### Location

- a. All test cases must be located in the directory `TestAutomation/testCases`

#### Name and type of the File

- a. File name: `testCase*.txt`
  - i. The asterisk represents a unique number, which will allow for identification of that particular test case
- b. File type: text
  - i. Only text (.txt) files will be interpreted as test cases and run accordingly.

### Structure of File Contents

- c. The following sections must be included in the following order, and on it's own line, for the test case to be used successfully:

Test Case ID: \*1\*

Requirement being tested:  
\*2\*

Component being tested:  
\*3\* - \*4\*  
Path to file: \*5\*

Method being tested: \*6\*

Test input(s) including command-line arguments: \*7\*

Expected outcome(s): \*8\*

Placeholder	Necessary Information	Notes
*1*	Test case ID number	Must match number in file name
*2*	Pertinent requirement	Each test case must be traceable to a requirement
*3*	Sugar Labs activity name	
*4*	Activity python file	
*5*	The path to the activity python file	This path will be from the home directory of the project formatted as "dir1/dir2/...dirn/file.py"
*6*	The method to be tested	Includes entire method signature
*7*	Parameters for the method call	
*8*	Expected outcome	

## Test Recording Procedures

To record the results of our tests, we aggregated the following information about each test case: test case identification, method signature, associated input, expected output, received output, and whether or not the test case passed or failed. This data was stored in a html report stored in a subdirectory of the TestAutomation directory. This test report will contain the following sections:

TestCaseID, Tested Method, Input, Expected Output, Actual Output, and Pass/Fail

The section TestCaseID will contain a hyperlink to the individual test case file, allowing for further viewing of information pertinent to the test case. See figure 4 (in chapter 4) for an example of the created report.

## Hardware And Software Requirements

There are no hardware requirements to use our automated testing framework other than a keyboard and monitor. As for software requirements, our automated testing framework requires a current version of Linux and for Python2.7 to be installed.

## Constraints

At the moment, there are no known constraints affecting the progress of our testing.

## Testing Schedule

We will be developing a testing framework over the course of this academic semester. We plan to meet progress deadlines on the following dates:

September 13

Checkout or clone project from its repository and build it, evaluating the project and experience so far.

September 27

Produce a detailed test plan, and specify at least 5 of the eventual 25 test cases for this project.

October 25

Submit an architectural description of our framework, full how-to documentation, a set of at least 5 of the eventual 25 test cases.

November 15

Complete the design and implementation of our testing framework as specified in Deliverable #3, and create 25 test cases that our framework will automatically use to test our H/FOSS project (Sugar Labs).

November 29

Design and inject 5 faults into the code we are testing that will cause at least 5 tests to fail, but hopefully not all tests to fail. Report the results of running the testing framework.

December 1

Submit a final report detailing all information covered in previous deliverables, including sections for our overall experiences, what we have learned and a self-evaluation.



## Chapter 3

### Automated Testing Framework

For this project, we designed a set of Python scripts that comprise an automated testing framework. This automated testing framework will iterate through a set of test cases and use the information within them to assess the Sugar Labs source code. These test cases (further described in chapter 4) dictate what methods in what component to test, along with any inputs and the expected outputs that should be given. The automated testing framework will run the specified methods with the given inputs and compare the expected results from the test case with the results it gets from the source code. The automated testing framework will then create a report containing all the information from the test cases and the results comparison. This report can then be used to determine how the code performed.

It is important to note that our automated testing framework can also test for the proper exceptions being raised. For example, dividing by zero results in a math domain error. By placing the error string (in our example: “Can not divide by zero”) the test cases can also test whether or not Sugar Labs handles exceptions in the proper manner.

### Automated Testing Framework Architecture

Our testing framework is composed of five python files (shown in figure 2) that interact with the source code and a set of test cases. These five files are:

[runAllTests.py](#)  
[parseTestCase.py](#)  
[runTestCase.py](#)  
[myExceptions.py](#)  
[progressBar.py](#)

The first file, `runAllTests.py`, comprises the main logic of our testing framework. It’s goal is to run every test case located in the `testCases` directory. `runAllTests.py` will then use `parseTestCase.py` to parse each test case (filtering out test cases that do not follow the naming convention) and collect all pertinent information (namely the test case ID, a path to a testable file, the method in that file to test, test inputs, and expected test outputs). With this information, `runAllTests.py` then uses `runTestCase.py` to run each test case. This will output the result of the tested method with the supplied input. `runAllTests.py` then compares this result to the expected output and determines if the test passed or failed from that determination.

The remaining two files, `myExceptions.py` and `progressBar.py`, add some utility to the automated testing framework. `myExceptions.py` defines a custom exception that is used as a means of ensuring durability. In the case where a `testCase` follows the naming convention, but does not follow the pre-defined template for its contents, a custom exception (`ImproperTestCaseSpecification`) is raised. This allows for errors due to improper structuring of

the test case (such as missing elements or superfluous elements) to be avoided. By raising this exception, the caller – runAllTests.py – knows to skip this test case. This prevents a poorly written test case from crashing the automated testing framework, and instead will allow the testing to continue. In a similar vein, an exception generated by running the test case will not crash the system either, as exception handling for runtime errors is built into runAllTests.py as well.

progressBar.py simply allows for runAllTests.py to show the user through the command line interface roughly how much of the testing is complete at any given moment. It does this by calculating how many tests have been run versus the total number of test cases that will be run. This percentage is displayed next to a progress bar that adjusts itself to the width of the console. After finishing every test case, the progress bar will display “Complete” to signify completion.

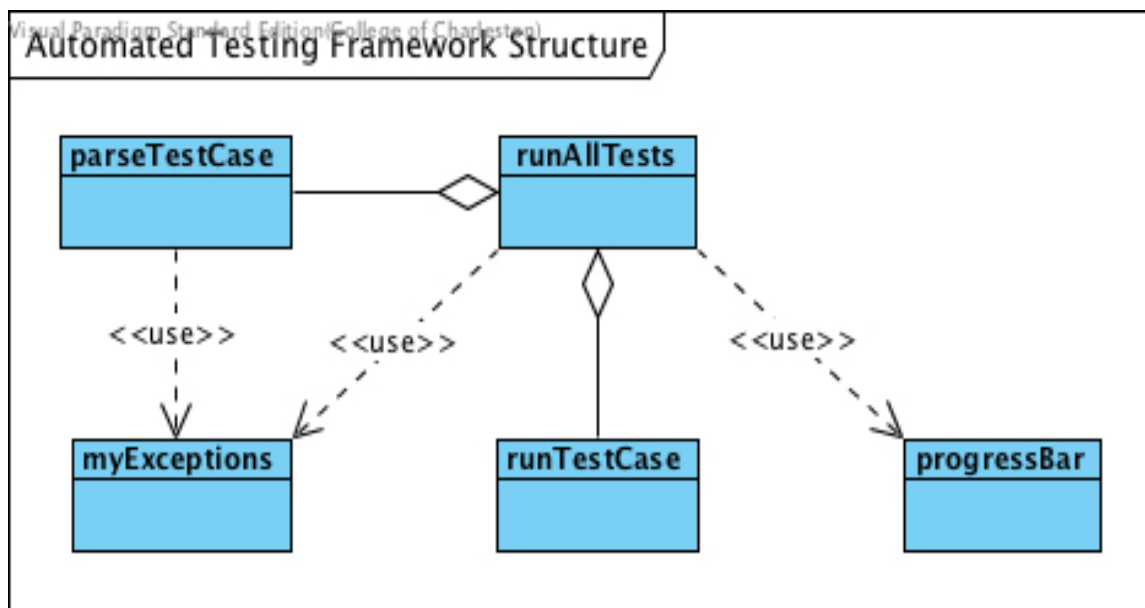


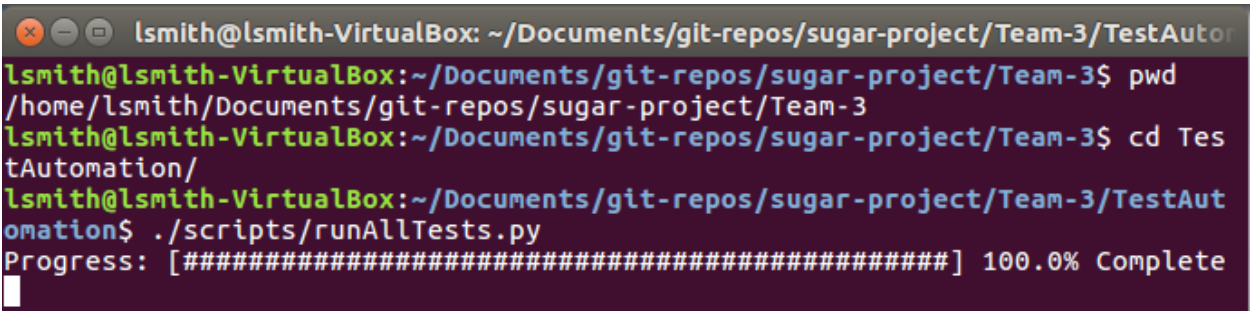
Figure 2: A succinct UML representation of the structure of our automated testing framework

## Using The Automated Testing Framework

To use our automated testing framework, you first need to checkout the code from our Team-3 Github repository. After this, our automated testing framework can be invoked by calling the script runAllTests.py from WITHIN the TestAutomation directory using the following command on the Linux command line:

```
./scripts/runAllTests.py
```

This will start the automated testing framework which will then iterate through every test case in TestAutomation/testCases, aggregate the results, and build a report of the results.



```
lsmith@lsmith-VirtualBox: ~/Documents/git-repos/sugar-project/Team-3/TestAutomation$ pwd
/home/lsmith/Documents/git-repos/sugar-project/Team-3
lsmith@lsmith-VirtualBox:~/Documents/git-repos/sugar-project/Team-3$ cd TestAutomation/
lsmith@lsmith-VirtualBox:~/Documents/git-repos/sugar-project/Team-3/TestAutomation$ ./scripts/runAllTests.py
Progress: [#####] 100.0% Complete
```

Figure 3: Invoking the automated testing framework

Test Case ID	Tested Method	Input	Testing Results		Pass/Fail
			Expected Output	Actual Outcome	
1	pow(a, b)	2, 10	1024	1024	Pass
2	add(a, b)	14.0, 7.5	21.5	21.5	Pass
3	mul(a, b)	15, -5	-75	-75	Pass
4	div(a, b)	120, 8	15	15	Pass
5	sqrt(a)	16	4.0	4.0	Pass

Figure 4: An example of the report generated by our automated testing framework

## Chapter 4

### Our Test Cases

The following test cases are the 25 test cases we have written for this project. With these 25 test cases, we attempt to verify that the calculator activity functions properly. To this end, we decided to test five functions: `pow(a, b)`, `add(a, b)`, `div(a, b)`, `sqrt(a)`, and `cos(a)`. For `pow`, `add`, and `cos` we proposed a wide scope to verify that the functions yielded correct results for all types of input. For `div` and `sqrt`, our focus was two pronged. Our goal was to verify that the functions yielded correct results for valid input values, but we also wanted to verify that the functions behaved properly when presented with certain invalid inputs. As an example, it is impossible to divide a number by zero, and when doing so the function should give us a divide by zero exception. In a similar vein, it is impossible to take the square root of a negative number (without using imaginary numbers). This is in line with how the two functions should perform, so we wanted to insure that not only would these two functions succeed where they should; they should also fail where they are expected to as well.

Test Case 1	
Requirement to Test	Correct and accurate exponentiation
Component to Test	<code>functions.py</code>
Method to be Tested	<code>pow(a, b)</code>
Input	2, 10
Expected Output	1024
Test Case 2	
Requirement to Test	Correct and accurate exponentiation
Component to Test	<code>functions.py</code>
Method to be Tested	<code>pow(a, b)</code>
Input	4.0, 2.5
Expected Output	32.0

Test Case 3	
Requirement to Test	Correct and accurate exponentiation
Component to Test	functions.py
Method to be Tested	pow(a, b)
Input	16, 0.5
Expected Output	4.0
Test Case 4	
Requirement to Test	Correct and accurate exponentiation
Component to Test	functions.py
Method to be Tested	pow(a, b)
Input	16, -0.5
Expected Output	0.25
Test Case 5	
Requirement to Test	Correct and accurate exponentiation
Component to Test	functions.py
Method to be Tested	pow(a, b)
Input	0, 5
Expected Output	0
Test Case 6	
Requirement to Test	Correct and accurate addition
Component to Test	functions.py
Method to be Tested	add(a, b)
Input	20, 5
Expected Output	25

Test Case 7	
Requirement to Test	Correct and accurate addition
Component to Test	functions.py
Method to be Tested	add(a, b)
Input	-5, -10
Expected Output	-15
Test Case 8	
Requirement to Test	Correct and accurate addition
Component to Test	functions.py
Method to be Tested	add(a, b)
Input	-40, 5
Expected Output	-35
Test Case 9	
Requirement to Test	Correct and accurate addition
Component to Test	functions.py
Method to be Tested	add(a, b)
Input	12.5, 5
Expected Output	17.5
Test Case 10	
Requirement to Test	Correct and accurate addition
Component to Test	functions.py
Method to be Tested	add(a, b)
Input	4.5, 7.25
Expected Output	11.75

Test Case 11	
Requirement to Test	Correct and accurate division
Component to Test	functions.py
Method to be Tested	div(a, b)
Input	120, 8
Expected Output	15
Test Case 12	
Requirement to Test	Correct and accurate division
Component to Test	functions.py
Method to be Tested	div(a, b)
Input	25.8, 3.0
Expected Output	8.6
Test Case 13	
Requirement to Test	Correct and accurate division
Component to Test	functions.py
Method to be Tested	div(a, b)
Input	28.4, -4
Expected Output	-7.1
Test Case 14	
Requirement to Test	Correct and accurate division
Component to Test	functions.py
Method to be Tested	div(a, b)
Input	-12, -6
Expected Output	2

Test Case 15	
Requirement to Test	Correct and accurate division
Component to Test	functions.py
Method to be Tested	div(a, b)
Input	120, 0
Expected Output	ERROR - "Can not divide by zero"
Test Case 16	
Requirement to Test	Accurate calculation of square roots
Component to Test	functions.py
Method to be Tested	sqrt(a)
Input	16
Expected Output	4.0
Test Case 17	
Requirement to Test	Accurate calculation of square roots
Component to Test	functions.py
Method to be Tested	sqrt(a)
Input	9.0
Expected Output	3.0
Test Case 18	
Requirement to Test	Accurate calculation of square roots
Component to Test	functions.py
Method to be Tested	sqrt(a)
Input	0.00000001
Expected Output	0.0001



Test Case 19	
Requirement to Test	Accurate calculation of square roots
Component to Test	functions.py
Method to be Tested	sqrt(a)
Input	0
Expected Output	0.0
Test Case 20	
Requirement to Test	Accurate calculation of square roots
Component to Test	functions.py
Method to be Tested	sqrt(a)
Input	-4
Expected Output	ERROR - "math domain error"
Test Case 21	
Requirement to Test	Accurate determination of cosine values
Component to Test	functions.py
Method to be Tested	cos(a)
Input	-3.141592653599790
Expected Output	-1.0
Test Case 22	
Requirement to Test	Accurate determination of cosine values
Component to Test	functions.py
Method to be Tested	cos(a)
Input	0
Expected Output	1.0

Test Case 23	
Requirement to Test	Accurate determination of cosine values
Component to Test	functions.py
Method to be Tested	cos(a)
Input	3.14159265359979
Expected Output	-1.0
Test Case 24	
Requirement to Test	Accurate determination of cosine values
Component to Test	functions.py
Method to be Tested	cos(a)
Input	6.283185307179590
Expected Output	1.0
Test Case 25	
Requirement to Test	Accurate determination of cosine values
Component to Test	functions.py
Method to be Tested	cos(a)
Input	12.566370614359200
Expected Output	1.0

## Chapter 5

### Fault Injection

For this segment of our project, we inserted faults into the source code for the purpose of seeing how it would affect our test cases. The goal was to insert faults that would make some, but not all, of our test cases fail and to observe the results. The following five faults caused eleven of our test cases to fail.

#### 1st Fault:

TestAutomation/project/sugar-calculate/functions.py

Line 351

Replace

```
return long(x) ** int(y)
```

With

```
return long(x) * int(y)
```

With this fault, we made it so that some test cases would fail depending on whether or not they reached this line inside a conditional statement. Specifically, this will occur if the first argument and second arguments off the add function are integers. This caused test case 1 to fail.

#### 2nd Fault:

TestAutomation/project/sugar-calculate/functions.py

Line 101

Replace

```
angle_scaling = ClassValue(1.0)
```

With

```
angle_scaling = ClassValue(2.0)
```

We added a fault to the variable `angle_scaling` in `functions.py`. This variable is assigned a value of 1.0 normally and it is used in the calculations of sine, cosine, and tangent values. Changing this value cased the calculation of our cosine values to be incorrect. This caused test cases 21 and 22 to fail.

#### 3rd Fault:

TestAutomation/project/sugar-calculate/rational.py

Lines 34 and 35

Replace

```
if n is not None:
    self.set(n, d)
```

With

```
if n is None:
    self.set(n, d)
```

We added a fault to the rational class so that it would skip some of the steps it takes upon initialization based on an erroneous boolean expression. This caused test cases 11 and 14 to fail.

**4th Fault:**

TestAutomation/project/sugar-calculate/functions.py

Line 413

Replace

```
return math.sqrt(float(x))
```

With

```
return math.sqrt(int(x))
```

This fault results in a loss of precision when using the square root function. Thus non-integers (that are not equivalent to whole numbers) will not calculate correctly due to the precision lost in the conversion. This caused test case 18 to fail.

**5th Fault:**

TestAutomation/project/sugar-calculate/functions.py

Lines 132-135

Replace

```
if isinstance(x, _Decimal) or isinstance(y, _Decimal):
    x = _d(x)
    y = _d(y)
    return x + y
```

With

```
if isinstance(x, _Decimal) or isinstance(y, _Decimal):
    x = _d(x)
    y = _d(y)
    return x + y
```

This fault results in the addition being performed only if the parameters for the function are instances of a decimal object. This caused test cases 6, 7, 8, 9, and 10 to fail.

# Chapter 6

## Our Experiences

Working with Sugar Labs has been both a remarkable and frustrating experience. Sugar Labs is an incredible piece of software with the potential to impact lives significantly, and as we worked with Sugar Labs, it was easy for us to see the good that it might do. That being said; at times it wasn't the easiest piece of software to work with. The documentation of the code was often lacking, hindering our efforts to use the software. In the beginning, we ran into numerous issues merely trying to compile the software with the tools provided by the developers. In order to get a functioning product, we were forced to disable a critical component of the software through various configuration files.

## What We Learned

This was a wonderful project to work on. Throughout this semester, our project afforded us many opportunities to develop and hone our skills in teamwork, interpersonal communication, and the usage of various collaborative tools (such as Github). For most of us, this was the most intensive project that we had participated in so far during our academic career. This project spanned the entire semester and required us to effectively coordinate and divide the workload from day one onwards. We also needed to learn how to use collaborative tools to maximize our efficiency. This meant that we needed to learn how to use Github, which most of our group had never used intensively before. This alone was an incredibly valuable skill to learn during the course, as version control software is something we will doubtlessly be using throughout our professional careers.

## Self-Evaluation

We feel that we have effectively worked throughout the semester and delivered a quality product, while meeting all deadlines. There are always things that can be improved upon. For example, we could certainly enhance our usage of Github. As we made changes to code and tried to push these changes to our Github repository, we found that we often encountered merge conflicts. This could have, in part, been avoided by using different branches as we developed certain features of the code. We will make sure to properly utilize this feature of Github going forward.

# Glossary

## **Automated Testing Framework**

an execution environment for automated tests that defines the format in which to express expectations, creates mechanisms to drive the application under testing, execute the tests, and report the results.

## **Github**

a web-based Git repository hosting service. It offers all of the distributed version control and source code management (SCM) functionality of Git as well as adding its own features. It provides access control and several collaboration features such as bug tracking, feature requests, task management, and wikis for every project.

## **Linux**

a Unix-like computer operating system assembled under the model of free and open-source software development and distribution.

## **One-Laptop-Per-Child**

a non-profit initiative established with the goal of transforming education for children in around the world; this goal was to be achieved by creating and distributing educational devices for the developing world, and by creating software and content for those devices.

## **Sugar Labs**

a software-development and learning community, which makes a collection of tools that learners use to explore, discover, create, and reflect. It distributes these tools freely and encourages its users to appropriate them, taking ownership and responsibility for their learning.. Helping learners all around the globe to "learn how to learn"