# Sugar Labs Automated Testing Framework

CSCI 362 Final Project

**Alex Skiff, Blaine Billings, Carson Barber, Chase Myers, Justin Willis**

A framework presented for the automated testing of the Sugar Labs software project

College of Charleston
CSCI 362: Software Engineering
Dr. Anthony P. Leclerc
30 November 2018

## Table of Contents

sugarlabs

# 0 Introduction

## 0.1 Project Description

Sugar Labs[1] is a free, open-source software that aims to inreases accessibility and ease of use for classroom-setting technology-based learning. Through the project, the developers hope to provide students with an easy to understand and fun learning environment for both the technological sciences and the basics of elementary education. Thier code consists of two main parts - a highly interactive and graphics-based front-end supported by a logical, decision-driven back-end that manages user data, journal logs, learning assignments, and all other relevant materials.

While the project's organization follows the software engineering design principle of "separation of powers", dividing up individual tasks among many disjoint pieces of code, the tests associated with each section are neither easily accessible nor comprehensive. This is the problem we aimed to tackle.

## 0.2 Testing Framework

Through our work, we aim to create an automated testing framework that both highly increases the coverage of unit testing, allowing for the verification and near guarantee of fewer program faults, and allows a user of the Sugar Labs software to be able to easily integrate their tests as well as provide additional testing input for existing test classes.

This document is organized as follows. In Section 2, we thoroughly detail how to properly install the Sugar Labs project, accounting for all nececssary software requirements and pre-installation command procedures. Section 3 introduces the testing plan, detailing the specifics of the five principal test cases that serve as an outline for the twenty-five introduced herein. In addition, in this section, we cover various requirements and restraints associated with such an undertaking so as to be prepared for whatever challenges may arise. In Section 4, we provide an architectural description of the automated testing framework, outlining the primary files and folders, as well as discuss in detail the implementation of the five initial test cases as introduced in the previous section. This serves as both a backbone of the testing framework and a guide for any future test cases. In Section 5, we include brief descriptions of all twenty-six

[1]https://github.com/sugarlabs

implemented test cases, omitting the specifics which can be found in the actual implementation and understood alongisde the models of those characterized in detail in the previous section. In Section 6, we inject faults into the project code so as to demonstrate the veracity of our framework before concluding with a final review.

## 1 Deliverable 1: Installation

### 1.1 Clone and Build

In order to obtain a local instance of the project, we accessed the **sugar** repository made by the **SugarLabs** project team on GitHub[2]. Using the command

**git clone sugarlabs/sugar**,

we were, then, able to clone the project to our local machines.

From there, we utilized a document written by the SugarLabs development team included in the root directory, **README.md**, to guide the process of building the project on the operating system we chose to employ, Ubuntu 16.04. This consisted of running the following commands:

```
sudo apt-get install sugar-* -y
aclocal
sudo apt-get install intltool libglib2.0-dev gtk+-3.0 -y
./autogen.sh
make
make install
```

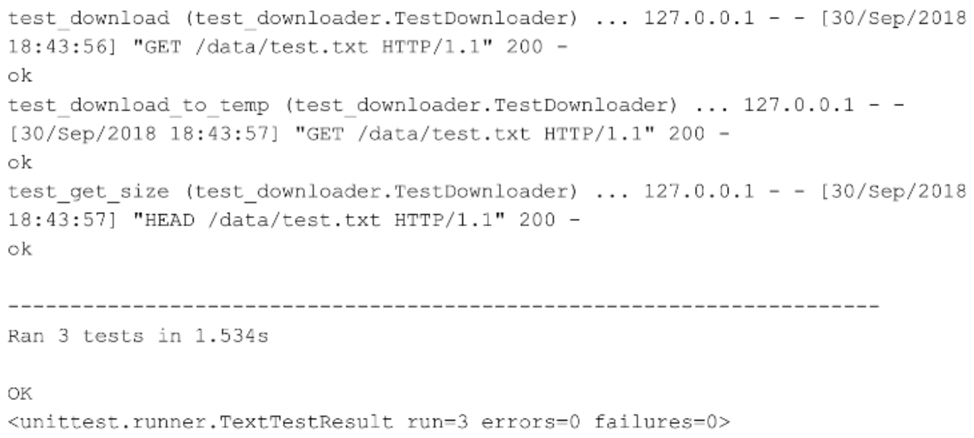
### 1.2 Existing Tests

SugarLabs provided multiple testing files for ease of use. After finding the files from the root directory, we searched Python's documentation for **unittest**, the imported package used for running the test cases. Through the use of this documentation, we were able to get most of the files tested with working output, an example of which is shown in Figure 1. The commands we used are as follows:

```
python
>> import unittest
>> import FILENAME
>> x = unittest.TestLoader().loadTestsFromTestCase(FILENAME)
>> unittest.TextTestRunner(verbosity=2).run(x)
```

Most of the tests passed, with only a few errors thrown. However, despite there being such files nicely laid out, there was neither explanation on how the tests are used nor documentation on what is and is not tested.

[2]https://github.com/sugarlabs/sugar



```
test_download (test_downloader.TestDownloader) ... 127.0.0.1 - - [30/Sep/2018
18:43:56] "GET /data/test.txt HTTP/1.1" 200 -
ok
test_download_to_temp (test_downloader.TestDownloader) ... 127.0.0.1 - -
[30/Sep/2018 18:43:57] "GET /data/test.txt HTTP/1.1" 200 -
ok
test_get_size (test_downloader.TestDownloader) ... 127.0.0.1 - - [30/Sep/2018
18:43:57] "HEAD /data/test.txt HTTP/1.1" 200 -
ok

----------------------------------------------------------------------
Ran 3 tests in 1.534s

OK
<unittest.runner.TextTestResult run=3 errors=0 failures=0>
```

Figure 1: A run of an included test file, test_downloader.py.

### 1.3 Team Evaluation

Overall the project is very organized with some help on building the project provided and only a little lacking information on testing what they have made. The developers were prepared for testing everything they needed to have tested, but only for themselves. It would have been more helpful for others looking in if there was a file that would automate the testing or at least more information on how to set up each individual test. We look forward to creating our own tests and exploring in more detail this project.

## 2 Deliverable 2: Introductory Test Plan

### 2.1 Testing Process

We have divided the process of our test plan for our five introductory test cases into the evaluation of three main subsystems. They are described as follows:

1. **Profile**: The **profile** subsystem of SugarLabs includes all files relevant to the creation of a user's profile, including the age, gender, etc. This will be evaluated based on input/ouput matching as well as on exception handling for invalid input.
2. **Journal**: The **journal** subsystem of SugarLabs includes all files relevant for the upkeep of a user's journal. Much like with the first subsystem, this will be evaluated based on input/output matching as well as on exception handling for invalid input.
3. **Activity**: The **activity** subsystem of SugarLabs includes all files relevant to tracking a user profile's assigned activities, essentially providing running assessment list for the user. This will be evaluated based on correctness of logged information with regards to simulated performed actions.

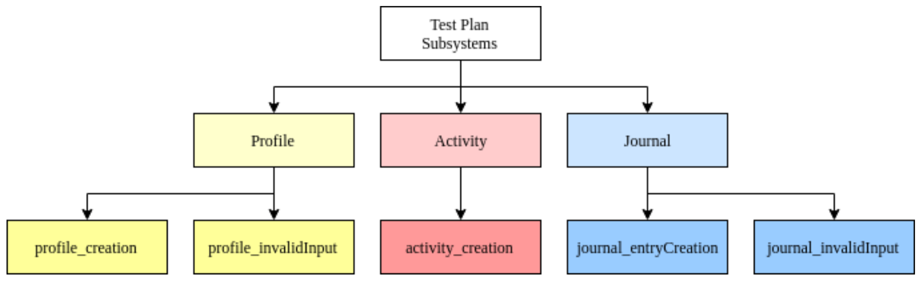


Figure 2: Overview of test plan subsystems and their associated test cases. (made using draw.io)

## 2.2 Requirements Traceability

For the aforementioned subsystems, the following files will be evaluated. This information is included for the ease of tracking the testing process and the evaluated files as well as the understanding of what processes in specific are being investigated.

1. **Profile**: colorpicker.py, agepicker.py, genderpicker.py for color, age, and gender selection, respectively, in the user profile creation process.
2. **Journal**: journalactivity.py, journalentrybundle.py, journalwindow.py for activity logging, entry bundling, and graphical output, respectively, in the user journal editing and viewing.
3. **Activity**: buddy.py for the activity assignment in the user activity view.

## 2.3 Tested Items

In this section, we described the specifc items mentioned in Section 2.2 and give an in-depth description of how they are to be tested through our framework. Figure 2 shows a visualization of three three general susbsystems and their associated test cases.

### 2.3.1 Profile

The testing of this subsystem and the respective colorpicker.py, agepicker.py, and genderpicker.py files will begin with the creation of a test_profile.py program with specific test_profile_creation and test_profile_invalidInput functions as test cases. These test cases go as follows:

1. **test_profile_creation**: This function will consist of the selection of a valid color and a gender and the input of a valid age. The test will be passed if profile creation is carried through with according input and without the throwing of an exception.
2. **test_profile_invalidInput**: This function will consist of the input of an invalid color, gender, or age. The test will be passed if profile creation with this input throws an exception.

#### 2.3.2 Journal

The testing of this subsystem and the respective journalactivity.py, journalentrybundle.py, and journalwindow.py files will begin with the creation of a test_journal.py program with specific test_journal_entryCreation and test_journal_invalidInput functions as test cases. These test cases go as follows:

1. **test_journal_entryCreation**: This function will consist of the opening of the journal entry window followed by the inputting of a valid journal entry name. The test will be passed if profile creation is carried through with according input and without the throwing of an exception.
2. **test_journal_invalidInput**: This function will consist of the opening of the journal entry window followed by the inputting of an invalid journal entry name (special characters, escaped characters, etc.). The test will be passed if profile creation with this input throws an exception.

#### 2.3.3 Activity

The testing of this subsystem and the respective buddy.py file will begin with the creation of a test_activity.py program with specific test_activity_creation function as a test case. These test case goes as follows:

1. **test_activity_creation**: This function will consist of the opening of the creation of a new activity and its assignment to a specific user profile. The test will be passed if activity creation is carried through with according input, assigned to the relevant profile, and without the throwing of an exception.

### 2.4 Testing Schedule

The tentative schedule for our testing plan has been defined as follows:

**11/9/2018**: Completion of automated testing framework
**11/19/2018**: Completion of design and implementation of testing framework; Evaluation of twenty-five test cases
**11/28/2018**: Completion of code injection of 5 faults; Testing of chosen twenty-five test cases with faulty code.

### 2.5 Test Recording Procedures

Each of the implemented test cases will be coded in python and run through the terminal as described in the first deliverable. For easy auditing and understanding of previous tests, this code will include the date, time, and current software version with all output automatically being written to a log file for that specific date. These will be aggregated in a **testing** folder in the root directory of this testing framework's project.

### 2.6 Hardware and Software Requirements

As of the time of this document being written, Sugar Labs requires the local operating system to be either Debian 0.110 or higher, Fedora, or Ubuntu 16.04 or higher. In addition, Python 2.7, the main codebase of the software, must be installed alongside the sugar-artwork, sugar-datastore, and sugar-toolkit-gtk3 packages which can all be obtained directly from the terminal. For building the project, the packages intltool, libglib2.0-dev, and gtk+-3.0 should also be installed on the system.

### 2.7 Constraints

Though there are not many constraints limiting the testing process of this project, we must plan to be punctual in every step of our testing framework's implementation as required by the general outline of this course. Due to the nature of this project (in it being carried out in an educational setting), we need not worry about budgeting, staff, or weekly meeting presentations, all of which are often necessitated in government or industry settings.

### 2.8 System Tests

The system tests will primarily consist of checking relevant software installations. The following software, shown with their according verification commands, will be used to check that the system is up-to-date and able to run SugarLabs:

- **Python**: python -v
- **intltool**: intltool --help
- **autoconf**: autconf -h
- **libglib2.0-dev**: dpkg --get-selections | grep libglib2.0-dev
- **gtk+-3.0**: dpkg --get-selections | gtk-3.0

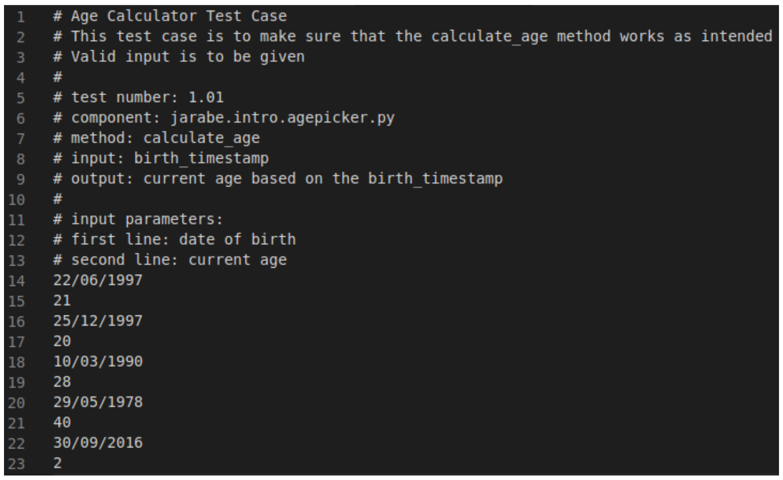
### 2.9 Team Evaluation

So far, our team seems to be making good progress. Weve been able to properly install and make the source code that we pulled off of GitHub. We split our first test cases into three subsystems: Profile, Journal and Activity. We plan to put our future tests into each of these subsystems to make testing easier and more streamlined.Our goal is to test files required for profile creation, such as picking an age and gender, as well as assigning journal activities.

## 3 Deliverable 3: Automated Testing Framework

### 3.1 Architectural Description

The automated testing framework has been subdivided in to following sections for ease of use:



```
# Age Calculator Test Case
# This test case is to make sure that the calculate_age method works as intended
# Valid input is to be given
#
# test number: 1.01
# component: jarabe.intro.agepicker.py
# method: calculate_age
# input: birth_timestamp
# output: current age based on the birth_timestamp
#
# input parameters:
# first line: date of birth
# second line: current age
22/06/1997
21
25/12/1997
20
10/03/1990
28
29/05/1978
40
30/09/2016
2
```

Figure 3: Example documentation for testCase input files, allowing users to be able to understand how to modify them as necessary.

↳**/TestAutomation**: Root directory for the automated testing framework
- ↳**/sugar**: Contains the cloned project files for the SugarLabs repository
- ↳**/scripts**: Contains scripts used for running a specific test case and for running all test cases
- ↳**/testCases**: Contains test case input
- ↳**/testCaseExecutables**: Contains the test cases written in python
- ↳**/temp**: Contains the output.log file where all testing output is piped
- ↳**/docs**: Contains the README.txt file for the automated testing framework
- ↳**/reports**: Contains all of the reports associated with the automated testing framework

## 3.2 Documentation

Each of the test cases have been written and saved to the /TestAutomation/testCaseExecutables folder, with their input files being located in /TestAutomation/testCases as described above. In each of these testCase input files is outlined a brief description of the test case and how one may go about modifying it for their own purposes. An example of such is provided in Figure 3.

## 3.3 Test Case Sepcifications

For the first step in the implementation of our automate testing framework we wrote and exectuted five primary test cases. These are specified below.

1. age_calculator
   - **Test ID**: 2
   - **Test Requirement**: Correlated age input is properly recognized by the system as correct input and subsequently processed.
   - **Test Component**: jarabe.intro.agepicker
   - **Test Method**: calculate_age
   - **Test Input**: Input is provided in the form of two-line input, the first being the birthdate of the user and the second the correct age of said user. Input: (22/06/1997, 21), (25/12/1997, 20), (10/03/1990, 28), (29/05/1978, 40), (30/09/2016, 2)
   - **Expected Output**: Output is expected to return True for each of the input test cases, as the calculated age and input age will match. Expected Output: True, True, True, True, True
2. age_calculator_invalid
   - **Test ID**: 3
   - **Test Requirement**: Noncorrelated age input is properly recognized by the system as incorrect input.
   - **Test Component**: jarabe.intro.agepicker
   - **Test Method**: calculate_age
   - **Test Input**: Input is provided in the form of two-line input, the first being the birthdate of the user and the second the correct age of said user. Input: (22/06/1997, 22), (25/12/1997, 21), (10/03/1990, 29), (29/05/1978, 41), (30/09/2016, 3)
   - **Expected Output**: Output is expected to return False for each of the input test cases, as the calculated age and input age will match. Expected Output: False, False, False, False, False
3. buddy_color
   - **Test ID**: 4
   - **Test Requirement**: A user must be able to set and be returned the color of their account's "buddy"
   - **Test Component**: jarabe.model.buddy
   - **Test Method**: set_color, get_color
   - **Test Input**: Input is provided in the form of a six-digit hex number, representing the buddy color. Input: FF00FF
   - **Expected Output**: Output is expected to return the input number for each of the input test cases, as the set method should properly set the color, and the get method should properly return the color. Expected Output: FF00FF
4. buddy_key
   - **Test ID**: 5
   - **Test Requirement**: A user must be able to set and be returned the key of their account's "buddy"
   - **Test Component**: jarabe.model.buddy
   - **Test Method**: set_key, get_key
   - **Test Input**: Input is provided in the form of a two eight-letter codes joined by an underscore, representing the buddy key. Input:

    DEADBEEF_DEADCODE
  - **Expected Output**: Output is expected to return the input key for each of the input test cases, as the set method should properly set the key, and the get method should properly return the key. Expected Output: DEADBEEF_DEADCODE

5. buddy_nickname
   - **Test ID**: 6
   - **Test Requirement**: A user must be able to set and be returned the nickname of their account's "buddy"
   - **Test Component**: jarabe.model.buddy
   - **Test Method**: set_nick, get_nick
   - **Test Input**: Input is provided in the form of a string, representing the buddy nickname. Input: bigboidan
   - **Expected Output**: Output is expected to return the input string for each of the input test cases, as the set method should properly set the nickname, and the get method should properly return the nickname. Expected Output: bigboidan

## 3.4 Team Evaluation

This deliverable was definitely the most challenging task so far for this project. We decided to use python as out scripting language since that is what the project was written in to begin with, making it much easier to interact with the project files the sugarlabs team created. After creating a few test cases, we had some other issues to resolve like redirecting the output from unittests to a file instead of the terminal. Overall we are happy with the framework layout and our ability to create the rest of the test cases moving forward.

# 4 Deliverable 4: Complete Testing Specifications

## 4.1 Documentation

As described in the previous section, the test cases have been implemented and saved into the directory structure as follows: each test case executable file has been saved in the /testing-framework/testCaseExecutables/ folder; each test case description has been saved in the /testing-framework/testCases/ folder; a short description as well as the inputs for the respective executable file have been documented in each of these text files.

## 4.2 Test Case Specifications

Below are detailed the specifications for all twenty-six test cases. Each brief description contains four main parts - the test case id number, the requirement being tested, the component being tested, and the method being testing.

1. **Activity Chooser**:
   - **Test ID**: 1

- **Requirement**: Checks ActivityChooser titles and constructor
- **Component**: jarabe.model.activitychooser.py
- **Method**: set_title

2. **Age Calculator**:
   - **Test ID**: 2
   - **Requirement**: Checks for valid timestamp calculation of a birthdate
   - **Component**: jarabe.intro.agepicker.py
   - **Method**: calculate_age
3. **Age Calculator Invalid**:
   - **Test ID**: 3
   - **Requirement**: Checks for invalid timestamp calculation of a birthdate
   - **Component**: jarabe.intro.agepicker.py
   - **Method**: calculate_age
4. **Buddy Color**:
   - **Test ID**: 4
   - **Requirement**: Checks for color change ability
   - **Component**: jarabe.model.buddy.py
   - **Method**: set_color
5. **Buddy Key**:
   - **Test ID**: 5
   - **Requirement**: Checks for key change ability
   - **Component**: jarabe.model.buddy.py
   - **Method**: set_key
6. **Buddy Nickname**:
   - **Test ID**: 6
   - **Requirement**: Checks for nickname changes
   - **Component**: jarabe.model.buddy.py
   - **Method**: set_nick
7. **Change Brightness**:
   - **Test ID**: 7
   - **Requirement**: Can set brightness levels
   - **Component**: jarabe.model.brightness.py
   - **Method**: set_brightness
8. **CMD Help**:
   - **Test ID**: 8
   - **Requirement**: Testing system command line
   - **Component**: jarabe.controlpanel.cmd.py
   - **Method**: cmd_help
9. **CMD Restart**:
   - **Test ID**: 9
   - **Requirement**: Testing the system command line
   - **Component**: jarabe.controlpanel.cmd.py
   - **Method**: node_restart
10. **Help Website**:
    - **Test ID**: 10

- **Requirement**: Checks for the link to the help website
- **Component**: jarabe.view.viewhelp.py
- **Method**: get_social_help_server

11. **HTTP Open**:
    - **Test ID**: 11
    - **Requirement**: Checks that http requests can be made
    - **Component**: jarabe.util.httprange.py
    - **Method**: open
12. **HTTP Size**:
    - **Test ID**: 12
    - **Requirement**: Checks that http requests are valid
    - **Component**: jarabe.util.httprange.py
    - **Method**: size
13. **Journal Comment**:
    - **Test ID**: 13
    - **Requirement**: Checks for journal comments
    - **Component**: jarabe.journal.expandedentry.py
    - **Method**: add_row
14. **String isASCII**:
    - **Test ID**: 14
    - **Requirement**: Checks to make sure strings can be represented in ascii code
    - **Component**: jarabe.desktop.keydialog.py
    - **Method**: string_is_ascii
15. **String isHex**:
    - **Test ID**: 15
    - **Requirement**: Checks to make sure the hex string comparison is working
    - **Component**: jarabe.desktop.keydialog.py
    - **Method**: string_is_hex
16. **String Hex Converter**:
    - **Test ID**: 16
    - **Requirement**: Checks the string hex converter
    - **Component**: jarabe.desktop.keydialog.py
    - **Method**: string_to_hex
17. **Friends Model**:
    - **Test ID**: 17
    - **Requirement**: Checks for the friends model creation
    - **Component**: jarabe.model.friends.py
    - **Method**: get_model
18. **Add Friend**:
    - **Test ID**: 18
    - **Requirement**: Checks for errors when adding a friend
    - **Component**: jarabe.model.friends.py
    - **Method**: add_friend
19. **Has Friend**:

- **Test ID**: 19
- **Requirement**: Checks for added friends to appear as added
- **Component**: jarabe.model.friends.py
- **Method**: add_friend

20. **String Normalization**:
    - **Test ID**: 20
    - **Requirement**: Tests string normalization (removing non-ascii characters)
    - **Component**: jarabe.util.normalize
    - **Method**: normalize
21. **Serial Number Generation**:
    - **Test ID**: 21
    - **Requirement**: Checks for randomized serial generation
    - **Component**: jarabe.model.session
    - **Method**: have_systemd
22. **System Session Support**:
    - **Test ID**: 22
    - **Requirement**: Checks for school session logging support
    - **Component**: jarabe.model.session
    - **Method**: have_systemd
23. **Get System Session**:
    - **Test ID**: 23
    - **Requirement**: Checks session logging
    - **Component**: jarabe.model.session
    - **Method**: get_session
24. **Set Volume**:
    - **Test ID**: 24
    - **Requirement**: Changes the system volume
    - **Component**: jarabe.model.sound.py
    - **Method**: set_volume
25. **Mute Volume**:
    - **Test ID**: 25
    - **Requirement**: Mutes the system volume
    - **Component**: jarabe.model.sound.py
    - **Method**: set_muted
26. **Speech Manager**:
    - **Test ID**: 26
    - **Requirement**: Creates a speech manager utility
    - **Component**: jarabe.model.speed.py
    - **Method**: get_speech_manager

## 4.3 Team Evaluation

In this deliverable our team had to finish flushing out all 25 of the test cases that our Testing Framework would be automatically testing as well as make any changes to the framework that we found necessary. We decided that the way the framework was set up was perfectly fine for the rest of the project. From here we had a large task ahead of us, planning out and executing 20 more test cases on top of the 5 that we had for the previous deliverable. We got off to a decent start knocking a few out pretty quickly and started to hit a bit of a wall. Our biggest problem was many of the classes and files this system used are graphical and required additional imports that were not needed for a successful build. That issue combined with the system being written in python, it made it very difficult to understand some of the files because even if we could get them imported for test cases, we were not sure what all of the parameters meant because of python being a dynamically typed language. We still managed to get all 25 test cases and some of them were able to test graphical components of the system without actually needing the graphics to appear on screen. Sugar Labs would is a great tool and idea, but with the lack of documentation and instructions it turned out to be much larger of a task to come in and get test cases written than we initially expected. Overall we are happy with what we have accomplished up to this point with the system, it would have just been much easier to understand Sugar Labs as a whole if the development team had more documentation throughout the code.

# 5 Deliverable 5: Testing Fault-Injected Code

## 5.1 Description

For this deliverable we have changed the code in multiple places inside of Sugar Labs system in order to make our testing framework fail on a few test cases. The changes could be removing method implementation, making mathematical mistakes and other changes to cause assertions to no longer be valid. This will help us further test out the framework to make sure that even with broken code, it does not cause our framework to crash

## 5.2 Documentation

When you first clone the directory, The full sugar labs source code will be in the testing-framework/sugar directory, the working sugar labs files that we have not changed will be in the testing-framework/scripts/fix/ directory, and the faulty files will be in the testing-framework/scripts/break/ directory.

To inject the faulty code, you will run the script breakSugar.py from within the scripts directory. After breaking the code, you will need to run the fix-Sugar.py script to get the original code back in the working sugar labs directory. Because of the sugar labs installation process, the code that is executed is actually found in the path: /usr/local/lib/python2.7/dist-packages/jarabe/.

When you run the fix or break scripts, we will be compiling the broken or fixed code, and replacing the files within this distribution folder. It is also possible that you will need to be root when you run the fix and break scripts.

## 5.3 Changed Code

The Sugar Labs software has been edited, as described in the preceding paragraphs, in the following source files with changes as described:

- **agepicker.py**: Changed the _SECONDS_PER_YEAR constant to $(365 * 24 * 60 * 60 * 60)$ instead of $(365 * 24 * 60 * 60)$ so our age calculation will no longer return the correct value, causing the age_calculator test case.
- **viewhelp.py**: Changed the social-help-server string to social-help in the get_social_help_server method, so our help window will no longer be able to find the correct help web page, making the Help Website test case fail.
- **httprange.py** Removed the line self._size = self._result from the size method. This will make the size of the web request always return 0, so the users would never be able to tell if the http requests are even working. This will also cause our HTTP Size test case to fail for the method size.
- **friends.py**: In the has_buddy method for a Friends object, we changed the check for buddy.get_key() in self._friends to not in friends. This makes the system think that people who are not friends are friends and vice versa. This will cause the has_buddy method fail within our Buddy Check test case.
- **schoolserver.py**: In the _generate_serial_number() we added a line to seed the random number generator so that this serial number creator is no longer random. In our test case we have the seed match what the system usually used, so now our Serial Number test case will fail

## 5.4 Team Evaluation

For this deliverable we have changed the code in multiple places inside of Sugar Labs system in order to make our testing framework fail on a few test cases. The changes could be removing method implementation, making mathematical mistakes and other changes to cause assertions to no longer be valid. This will help us further test out the framework to make sure that even with broken code, it does not cause our framework to crash. This wasn't too difficult to plan out, the hardest part was replacing the actual sugar labs code. After installation Sugar Labs installs everything to the python distributable folder on your computer with the compiled files. To accompany these installations, we wrote scripts files to compile our changed code and replace the files in the sugar install directory. We also wrote a script file to replace the original code.

# 6 Overall Experiences