TERM PROJECT FINAL REPORT

CSCI 362

Team Soft Boys

Joe Ayers, Maz Little, Jasmine Mai, Joe Spencer

Fall 2017

Date: 11/28/17

TABLE OF CONTENTS

Introduction

This project was completed over the course of the Fall 2017 semester for Software Engineering (CSCI 362). At the beginning of the semester, we were assigned the task of developing an automated testing framework for a pre-existing Humanitarian Free and Open Source Software project (HFOSS). Over the course of the semester, we narrowed down our HFOSS project choices, developed a testing plan, completed the automated framework, and injected faults into our code to assess whether or not the framework would catch the faults. This project depended heavily on team collaboration and effective planning as much as it did on our coding ability, mirroring our previous experiences and future expectations of working in the industry. The following report details our experiences completing the project as a narrative, and includes code samples, instructions on how to run our completed automated testing framework, and our assessment of the project and our success.
The formal goal of this project is given in the specification document as follows:

- “The testing framework will run on Ubuntu (Linux/Unix).
- The testing framework will be invoked by a single script from within the top level folder using “./scripts/runAllTests.(some scripting extension)” and will access a folder of test case specifications, which will contain a single test case specification file for each test case.
- Each of these files will conform to a test case specification template that you develop based on the example template below.
- Each test case specification file contains the meta-data that your framework needs to setup and execute the test case and to collect the results of the test case execution.”

There also were a specified framework directory structure, file naming convention, and automated report outlined in this document. The below report serves as our record of completing these specifications.

Chapter 1: HFOSS Project Evaluations

Initial Options

We were tasked with choosing our top choices for an HFOSS project to test throughout the course of this semester. Our top five, including brief descriptions, are detailed below:

- Amara, a video captioning and translation service. The key benefit of Amara is to make important videos more accessible to a wider audience (the hearing impaired, people with language issues, etc.)
- Tanaguru, visibility and contrast utility
- Martus, an open-source encryption software
- FBReaderJ, a text-to-speech tool for use with bookshare
- Glucosio, an open-source resource for diabetes research

First Project

We initially chose the Tanaguru contrast finder. However, we ended up choosing not to use this project as it was documented poorly, and we found it too simple to be able to break 25 test cases down from the code. Additionally, we had difficulties getting the program to install and run on our machines.

Final Selection

After reevaluating Tanaguru, we chose instead to work on Sugar Labs. It was not one of our original options, but we thought given its extensive documentation, years of existence, and large dev team that it would be better suited for adapting to our project. Sugar has a step-by-step side that breaks down how to install the different builds for different systems, located on their developer documentation pages. As the page suggests for testers, we set up a Sugar Live Build environment on a VirtualBox virtual machine.

Chapter 2: Test Plan

Team Structure

In order to best divide and complete this project, we came up with a plan based on readings from class and team experience with project management. Each team member was responsible for a portion of the final product, although no team member worked on any deliverable alone. Instead, each member was designated as a "lead" for a section of the project.

The team responsibilities are detailed below:

All team responsibilities	Be able to run environment on their machine Clerical/formatting/documentation for assigned deliverables Contribute to completion of all deliverables
Lead responsibilities	Plans deliverable component schedule, including sessions to work on code Assigns team members tasks and follows up Has final say on decisions made about deliverable
Joe Ayers - Scripter	Lead on runAllTests script Creates final presentation
Maz Little - Admin	Lead on Testing Report Compiles Final Report PDF
Jasmine Mai - Framer	Lead on Test Case Template Designs final poster
Joe Spencer - Tester	Lead on Test Cases Compiles final GitHub/Wiki/Blog downloading and formatting

Test Plan

At the beginning of the project, we intended to start by testing top tier components, such as the activities' ability to start and stop, and work our way down to more basic components, such as the system's ability to store the user's age.
The original methods we chose to test were:
- launch_and_stop_activity
- calculate_birth_timestamp
- Test_webservice

In addition to the delegation of work for each activity, we also made a more detailed plan surrounding when we wanted deliverables and components due for each deliverable (bolded items indicate a due date designated in the syllabus):

- **10/3 - finished test plan with five general test cases**
- **10/31 - first test framework**
    - rough ideas for 25 test cases
    - test case template (Jasmine)
    - testing report (Maz)
    - runAllTests script (Joe A)
    - 25 test cases (draft code) (Joe S)
    - 10/12 - 10 test cases
    - 10/19 - 15 test cases
    - 10/26 - 20 test cases
- 11/7 - refine test framework
- 11/9 - compile test reports
- **11/14 - finished and implemented test framework with 25 test cases**
- 11/16 - final pass at test framework
- **11/21 - fault testing**
- 11/21 - presentation finished; final report first draft
- 11/21 - poster draft finished
- 11/27 - review final report draft
- **11/28 - final report**
- **11/30 - final report presentation**

Chapter 3: Framework Progress

Framework Progress

After some development of our framework, we found that testing the methods outlined in our original test plan would only test the reading and writing of information entered by the user, and not anything useful for a software developer to know (beyond whether the system could read strings it stored). We then re-selected test cases that would better fit what we wanted to test. For this purpose, we decided to test some of the mathematical functions that the OS used in many of its activity and operating system processes.

At this stage in the project, we reported the following on our framework:

> "Currently, we have a framework that can run specified inputs and output print statements regarding the status of the test to an unformatted html file that opens automatically. The framework script locates the testing report file, empties it, navigates to the test case folder, runs the test case files in the folder, outputs each result to the HTML file, and opens it. … We currently have our test cases hard coded into another python file to ... make sure our test frame works; the file loops through a pre-defined list of inputs and feeds them to the above code (rather than opening individual files). Our next steps with this script are to define (twenty more) individual inputs and refine the script so that it all runs automatically."

Test Case Identification

We identified the following methods as those we chose to test:

- Factorize (finds the prime factors of a number)
- B10bin (binary conversion)
- Int (returns if an input is an int)

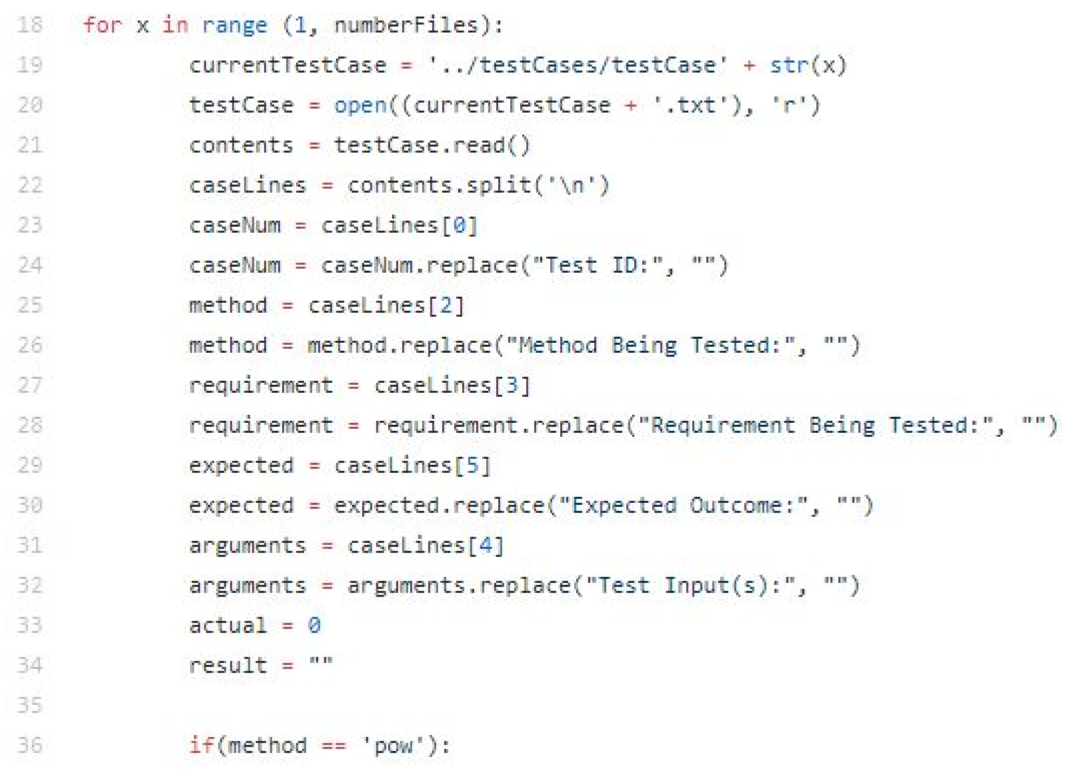
Chapter 4: Completed Test Framework

Completed Framework

Our test script is now written in Python, and takes in and parses the information in the text case files to run the test cases automatically, including outputting a formatted HTML table.

Code Excerpt

The below is an excerpt from our runAllTests.py script. It includes the beginning of the automated loop set up with the pre-determined number of test case files, which finds the next test case file, generates the beginning of the HTML file, and sets up the information required to run the rest of the test. At the bottom of this screenshot is the beginning of the logic to assert which method the test case is testing, then parses the inputs for that particular function.



```
18  for x in range (1, numberFiles):
19          currentTestCase = '../testCases/testCase' + str(x)
20          testCase = open((currentTestCase + '.txt'), 'r')
21          contents = testCase.read()
22          caseLines = contents.split('\n')
23          caseNum = caseLines[0]
24          caseNum = caseNum.replace("Test ID:", "")
25          method = caseLines[2]
26          method = method.replace("Method Being Tested:", "")
27          requirement = caseLines[3]
28          requirement = requirement.replace("Requirement Being Tested:", "")
29          expected = caseLines[5]
30          expected = expected.replace("Expected Outcome:", "")
31          arguments = caseLines[4]
32          arguments = arguments.replace("Test Input(s):", "")
33          actual = 0
34          result = ""
35
36          if(method == 'pow'):
```

An excerpt from our runAllTests.py script, lines 18 - 36.

Test Case Changes

We decided to not use all of the test cases we came up with for the last deliverable, as they only test getters and setters, and nothing the operating system used in calculations or did anything other than store data. They helped us learn more about the way Sugar is set up and the difficulties inherent in testing Sugar (mainly the dependencies), but ultimately do not reveal a lot about the system and aren't very complex tests. To that end, we've removed the sugar-master folder from our repository.

Instead, our test cases now come from some of the more complex math functions in the operating system that some of the physics and math activities rely on. These include a factorizing function that searches for the prime factors of a number, a function that takes in a binary number and converts it to a decimal number, a function that does the opposite, and a function that determines whether or not a number is an integer. Each of these functions has the same rough spread of inputs: Null; Empty; A character string; 0; A non-zero integer that satisfies the test conditions; A non-zero integer that does not satisfy the test conditions (if applicable)

We selected this testing scheme to be able to cover the entire spread of input options possible, to ensure that the functions can handle any input. The remaining functions that these activities perform are handled by the PyGame library, which is already rather robustly tested.

Biggest Difficulties

First, the way sugarlabs is setup is riddled with dependencies. This was a continual problem we had to grapple with, including having to set up dummy files to make Sugar run.

Second, this compounds another problem: there are several libraries that need to be imported for some of the functions we're testing, but some of the imports weren't recognized when we ran the script (it was these files, specifically, that we needed to mock to get Sugar to run).

Finally, on the more conceptual level it was difficult to decide exactly what we want to attempt to test about sugar labs, since it is an entire operating system. We couldn't (and weren't) aiming to produce an entirely comprehensive testing frame, and as a result waffled back and forth on whether to test just the core operating system, or include some of the activities.

Biggest Successes

First, this success may seem obvious, but we did complete a functional testing framework to the specifications given in the document at the beginning of the semester. Projects in the industry often are canceled, delayed, or abandoned. All of these outcomes were still possible despite this project being an academic assignment, but we were able to complete what was most of our first development project using someone else's code.

Next, this narrative document belies how many times we had to adjust what we were doing for our project in order to complete the project as specified. However, change management is an important part of every project, and this team handled change well. We made a series of decisions about what we wanted to test in the interest of learning more about testing a complex system such as an operating system, and documented the rationale behind each decision. These decisions often came out of unforeseen problems, such as the relevance of the methods we initially chose to test to the system, or simply a lack of project documentation.

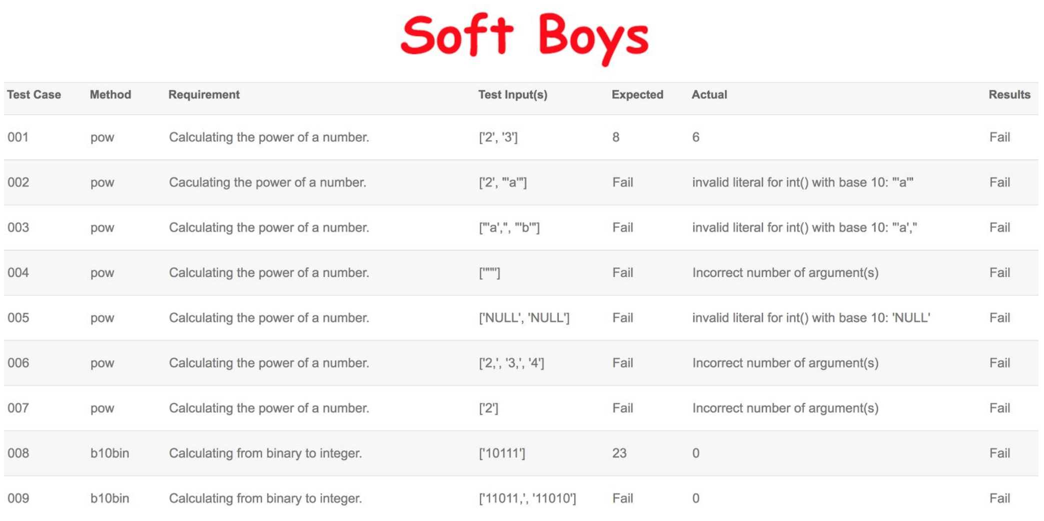
Lastly, we functioned well as a team. There's very little guarantee of a team clicking, even if (or especially if) they are already friends, but we all felt good about working together and the work we produced. Most importantly, we came out of this experience without hating each other, and ready to work on a similar collaborative project again.

Framework Explanation

Sugar Labs is essentially an operating system that is able to run different packages depending on the software. Therefore, we tested a package that works coincide with Sugar Labs which is mostly mathematical functions. The operating system, Sugar Labs, has multiple dependencies in order for the software to operate. The system was behind on its dependencies so it was not functioning fully, instead we decided to work on a package in Python.
The structure of our framework, as required, relies on folder structure. It is a shell method that runs with a set of inputs and is method- or test-case-agnostic. The script is able to take inputs from text files located in a test case folder. In `../Soft-Boys/TestAutomation/testCases` each test case has a method with a dedicated driver in the `../Soft-Boys/TestAutomation/testCasesExecutables` folder.

- How to download and install sugar
    - (Install and) Create a new instance of a virtual machine
    - Follow the instructions located here on how to install an instance of Sugar Live Build. Note that you will need to download 1.3 GB of data in order to install this system.
- How to download and set up framework
    - Navigate and clone our github: https://github.com/CSCI-362-02-2017/Soft-Boys on your VirtualMachine
- How to run framework
    - Locate to `../Soft-Boys/TestAutomation/scripts` from your terminal
    - Run the script with the following line in your terminal `python ./runAllTests.py`
    - The script will finish and produce a report located in the `../Soft-Boys/TestAutomation/reports`

A screenshot of our final HTML report.



Soft Boys

Test Case	Method	Requirement	Test Input(s)	Expected	Actual	Results
001	pow	Calculating the power of a number.	['2', '3']	8	6	Fail
002	pow	Caculating the power of a number.	['2', "'a'"]	Fail	invalid literal for int() with base 10: "'a'"	Fail
003	pow	Calculating the power of a number.	["'a',", "'b'"]	Fail	invalid literal for int() with base 10: "'a',"	Fail
004	pow	Calculating the power of a number.	["''"]	Fail	Incorrect number of argument(s)	Fail
005	pow	Calculating the power of a number.	['NULL', 'NULL']	Fail	invalid literal for int() with base 10: 'NULL'	Fail
006	pow	Calculating the power of a number.	['2,', '3,', '4']	Fail	Incorrect number of argument(s)	Fail
007	pow	Calculating the power of a number.	['2']	Fail	Incorrect number of argument(s)	Fail
008	b10bin	Calculating from binary to integer.	['10111']	23	0	Fail
009	b10bin	Calculating from binary to integer.	['11011,', '11010']	Fail	0	Fail

Chapter 5: Fault Injection

Fault Choice

It was relatively easy for us to assess the functions we selected for our test cases, and determine the best places to insert faults. We tried to approach fault insertion by making changes that could conceivably be the result of human error, rather than trying to come up with large, structural errors. It helped that the functions we are testing fall on the simpler side. At this point in the project, the code and script are familiar enough to us that this was a relatively simple task to fulfill after walking through the code in a group.

As stated earlier, we decided to move on from testing the simple functions that Sugar uses to more complicated logic utilized by the sugar OS. These included functions such as factorize, binary conversion, and determining if a number is an integer. However, when we went into the code again to assess it for possible fault injection sites, we found that the logic still wasn't complicated enough for an injected fault to feel like it actually had an impact. To that end, we modified and added to our test case list, so that we now are testing the below methods.

Tested Methods

We chose the below methods as our final set to be tested:

- Pow
- Factorize
- Factorial
- Inverse
- B10bin

Fault Breakdown

The below is a breakdown of where code was changed for each function, and why.

- Pow
    - A function to raise a specified number to a specified power
    - The function included a line with a double asterisk for multiplication; we removed one asterisk at line 418
    - We already have test cases for this function
- Factorize
    - A function to find the prime factors of a number
    - The function included a line where the number passed to it was divided modularly; we changed this modular division to regular division at line 291
    - We already have test cases for this function
- Factorial

    - A function to calculate the factorial of an integer
    - We removed the check to make sure the number is less than 0 at line 261, and inserted a test case for a negative number.
    - Test cases:
        - Null
        - Empty
        - 0
        - -1
        - 4
- Inverse function
    - A function to invert a number passed to it. I.e., if the function is passed '4,' it returns ¼
    - We edited the comparison to assert whether or not the numerator of the number passed to this function is 0 (as if it is, the inverse will be impossible). To that end, we changed the comparison to another, wrong comparison (== to >) at line 319
    - Test cases
        - Null
        - Empty
        - 0
        - Abc
        - 0.5
        - 25
- B10bin
    - A function to convert a decimal number to a binary number
    - In the function, there is a comparison to make sure that the value passed is bigger than 0. We changed the comparison from >= to > on line 188
    - We already have test cases for this function

Chapter 6: Project Self-Assessment

Assignment Evaluation

General suggestions

The biggest improvement we could suggest for these assignments is to provide further details on what should be included in each deliverable. The final product requirements are outlined in detail, but for people with little to no experience planning large, formal projects (which is what students are), it is difficult to break down how and when parts of a project need to happen, and what should be included in each deliverable.

One solution we suggest for this problem is, possibly for at least the first deliverable or two, to have a suggested report outline. This document could look much like the Team Projects Specification document, where the components of what should be in the final product are outlined. For example, it might include a list of suggested report sections, and a brief sentence or two on what should be included in each section.

D1: Evaluate HFOSS projects and select a project to test

This assignment could benefit from a brief overview of what projects students have chosen in the past, and why. It could be as simple as compiling the best Deliverable #1s from the past few years and presenting a few of them as an example, or as complex as a guest speaker who took this class and talks about their experiences over the course of the entire semester.

D2: Create a test plan

We discussed project management a moderate amount in this class, but this deliverable seems the most like it could benefit from some more focus on planning. We found that the effort put into defining team roles from the beginning, and what each member was required to complete or help on, was a significant boost to productivity. As mentioned in the Mythical Man Month article, increasing the amount of people on a project (as we did by including a fourth member) would not necessarily translate to a product being produced faster. With that in mind, designating one person to be in charge of each part of a project (a “lead”), who would keep track of that aspect of the project and delegate work to the other members for it, made sense. Every single member still worked on the project, but we were better able to divide work individually and meet with work and ideas with this organizational structure. This isn’t necessarily the solution for every team, but spending more time extrapolating how a good dev team is set up from the implications of the Mythical Man Month article seems like it could be worthwhile for classes in the future.

D3: Testing framework progress

During at least a third of the classes this semester, we were given time in class to meet with our groups, and possibly meet with our professor during that time if we had concerns. This deliverable was important in terms of having a concrete due date to have progress by for a long-term project. However, given that this is an “in-progress” due date, it may be more prudent to have this deliverable be more informal. Instead of a demo and presentation, we suggest possibly having the classes on this week be time to meet individually with the professor in a separate room for a longer period of time, while the rest of the class is given time during scheduled hours to work together. This both guarantees at least some in-person collaboration time, and also gives groups a chance to more informally work through problems with their projects with someone with more experience.

D4: Completed test framework

This deliverable was rather straightforward and didn’t seem to need any improvements.

D5: Fault injection

This deliverable was also relatively straightforward at this point in the semester.

Self-Evaluation

This project was definitely challenging because developing test cases for an open source project that isn’t our own entailed having to break down and understand the system first. In fact, our biggest changes came from having to reevaluate our project based on what we learned about the system after testing it a bit.
Part of what made this project a success for us was the amount of planning we put into it, which enabled us to better follow through on the work required to finish the project. It was a lot easier for every group member to be accountable for the work when we defined what everyone was responsible for ahead of time. Not only did we use the organizational structure outlined above, but we also relied on a piece of team collaborative software (called Band, a product of the same company that produces Line, the most popular messaging service in Japan). Between our organization, open communication, and access to collaborative and organizational platforms, the project was much more smoothly executed than previous group projects we’ve worked on.