College of Charleston - CSCI 362 - Dr. Bowring

HFOSS Project Final Report

Austin Hunt & Bryce Charydczak

Contents

Introduction

The following is an extensive outline of our experience with planning and developing a testing framework for our chosen HFOSS project (Amara). The project, on which Dr. Bowring's CSCI 362 is largely centered, allows for the software engineering concepts covered in readings and in class to extend from theory into application, and it illuminates the vital importance of well-designed tests when ensuring the dependability of large pieces of software.

Chapter One

Part One: mWater

We initially chose to explore the HFOSS project *mWater,* a free Android application that enables users to monitor, analyze and report water quality information. The app, via image processing, allows one to observe the presence of bacteria in incubated water samples and report this data to a central server, and this data is then used to create a map based on regional water quality that is viewable by all app users. We chose the project for multiple reasons, the primary ones being that 1) it was written in Java (a language we were familiar with), 2) the repository contained a directory filled with test files, and 3) water quality is something we recognized as universally significant.

Problems with mWater

After cloning the GitHub repository (found at **github.com/mWater/mWater-Android-App**) and trying to build it with Netbeans, we realized that the project contained many dependencies on the Android software development kit. In an attempt to resolve these dependencies, we installed Android Studio, the development environment based on IntelliJ IDEA for Android application development. Now in Android Studio, we attempted again to build the project but ran into additional problems which prevented the project from building (for example, some missing project files created more dependency issues). After searching the GitHub Wiki for solutions to the problem, we found that none of the links provided by the developers were functional, and we then realized that the mWater application we were working with had been superseded by a new version (found at **github.com/mWater/app-v3**). Moving on to this new version, we were disappointed to find that it was written completely in CoffeeScript, a programming language that compiles into JavaScript which neither of us were familiar with. On the other hand, we discovered that the project did include a test suite

containing a set of 15 different tests, so we each spent an hour learning the syntax of the new, fairly simple language in order to gain a basic understanding of the new project version. Unfortunately, after opening the cloned project in the Netbeans IDE, we were again unable to build the project. Looking for instructions on the GitHub page about fixing the problem led us to find a walkthrough in the repository's README file. It comprised a series of 8 steps, requiring the installation of the following:

- Node.js (JavaScript runtime environment)
- Grunt-cli (grunt command-line interface)
- Browserify (development tool for writing node.js style modules that compile for use in browser)
- Bower (package manager for the web)
- Cordova (mobile app development framework for developing apps with non-platform-specific APIs)

On account of neither of us being able to finish the provided walkthrough due to independently-encountered installation problems (combined with our new responsibility of adapting to CoffeeScript and the very apparent lack of sufficient documentation to reference), we decided to switch to a more fully-documented (and more *testable*) HFOSS project: Amara.

Part Two: Amara

We initially designated Amara as our second HFOSS project candidate because we were both highly interested in the process of video-subtitling, and we switched to it because it became clear that building the project and developing tests for it would be much less challenging in light of its much more extensive documentation. Also, the fact that the latest push to the repository- found at **github.com/pculture/unisubs** - occurred less than a year ago (as opposed to 3 years ago) made it clear that this was not a forgotten project, and thus problems should be less likely to arise. We quickly found that the README in the project repository included a detailed, step-by-step explanation of how to get started building the project[1].

Amara: Getting Started

---

[1] "GitHub - pculture/unisubs: Amara." https://github.com/pculture/unisubs. Accessed 30 Sep. 2017.

Amara uses Docker, an open-source tool designed to ease the challenge of creating, deploying and running applications via the use of containers (i.e. standalone, executable packages of software that contain everything they need to run). Neither of us were familiar with Docker prior to this project, but we quickly learned after some initial research that containerization (which Docker provides) of software is highly beneficial (and increasingly popular) in that it allows for the creation of easily portable, software-defined environments that are abstracted from the host system. In the context of testing, a concept which this project is centered on, containers allow for greater confidence that software failures won't occur after deployment because the application is *tested* in the same environment that it is eventually *run* after being deployed (i.e. inside the container).

After cloning the project to our computers, the next step in the provided walkthrough was to install docker-compose (a tool for defining/running multi-container applications). This installation, however, depended on the pre-installation of Docker Engine.

This process is illustrated by the following commands:

```
$ sudo apt-get update

$ sudo apt-get install \
    linux-image-extra-$(uname -r) \
    linux-image-extra-virtual
```

1) Fetch new versions of packages existent on the machine

2) Install the linux-image-extra-* packages to enable Docker to use **overlay2** storage driver (used to manage images and layers on Docker)

Install Using Docker Engine Repository (Reference: "Get Docker CE for Ubuntu")[1]

1) Setting Up the Repo

    a) [2]Update **apt** package index

```
$ sudo apt-get update
```

    b) Install packages to allow **apt** to use repository over HTTPS

---

[2] "Get Docker CE for Ubuntu | Docker Documentation." https://docs.docker.com/engine/installation/linux/docker-ce/ubuntu/. Accessed 30 Sep. 2017.

```
$ sudo apt-get install \
    apt-transport-https \
    ca-certificates \
    curl \
    software-properties-common
```

c) Add Docker's official GPG key

```
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
```

d) Set up the stable repository

```
$ sudo add-apt-repository \
    "deb [arch=amd64] https://download.docker.com/linux/ubuntu \
    $(lsb_release -cs) \
    stable"
```

2) Install Docker
    a) Update the **apt** package index

```
$ sudo apt-get update
```

    b) Install latest version of Docker Community Edition

```
$ sudo apt-get install docker-ce
```

    c) Verify the installation by running the **hello-world** image

```
$ sudo docker run hello-world
```

The following terminal screenshot displays the verification which appears following the previous command.

```
austinhunt@austinhunt-VirtualBox:~$ sudo docker run hello-world
[sudo] password for austinhunt:
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
5b0f327be733: Pull complete
Digest: sha256:b2ba691d8aac9e5ac3644c0788e3d3823f9e97f757f01d2ddc6eb5458df9d801
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.
```

Install Docker Compose

 With Docker Engine installed, Docker Compose can now be installed with the following command:

> **sudo curl -L https://github.com/docker/compose/releases/download/1.16.1/docker-compose-`uname -s`-`uname -m` -o /usr/local/bin/docker-compose**

In order to execute the file, executable permissions need to be applied with the **chmod** command, as follows:

```
sudo chmod +x /usr/local/bin/docker-compose
```

With Docker Compose installed, the Amara docker image can be built. An image, as eloquently described by **paislee** on Stack Overflow[3], is a static, immutable file which is essentially a template for a container; in fact, an image *produces* a containers when started with **run.**

Before building the Amara docker image, one must first ensure they are in the unisubs directory. For example, we cloned the unisubs repository to the Desktop, and then **cd**'d into the unisubs directory, as shown below:

git clone git://github.com/pculture/unisubs.git unisubs

cd unisubs

---

[3] "Docker image vs container - Stack Overflow." 19 May. 2014, https://stackoverflow.com/questions/23735149/docker-image-vs-container. Accessed 30 Sep. 2017.

To build the docker image, the following command is used:

```
sudo bin/dev build
```

The next step is to configure the application database. In order to do this, the following command is used:

```
sudo bin/dev dbreset
```

Once the database is configured, the Amara containers can be started with:

```
sudo bin/dev up
```

The next step is to modify the local machine's *hosts* file. This Ubuntu-based plain text file is designed to map human-friendly hostnames to numeric IP addresses (which identify the host in an IP network). The file contains two fields: the first is for IP addresses, the second is for hostnames. In this case, we want to map the hostname unisubs.example.com to the IP address 127.0.0.1 (that of the local host). To modify the hosts file, the following command is used:

```
sudo nano /etc/hosts
```

Once inside the file, the line

```
127.0.0.1        unisubs.example.com
```

should be added underneath the hostname-IP mappings that are already there, with the two field entries separated by a tab. The entry must then be saved, and the file can be closed. The purpose of adding this line to the hosts file is to ensure that Twitter and Facebook open authorization (**oauth**) works properly. That is, Amara user account information can be used with the third-party services Twitter and Facebook without exposing the Amara user's password. To ensure that the added mapping works properly, the site can be accessed at **http://unisubs.example.com:8000.**

Part Three: Developing and Deploying a Dynamic Web Platform to Enhance Teaching in the Field of Computer Science

After iteratively developing upon the Amara architecture we deemed it to be too demanding of a task to become articulate with and develop tests for the architecture. Amara taught us serious lessons on familiarizing with a framework and developing for the HFOSS project. The limitations we found were that of deciphering the given code and trying to find ways to develop tests for it. Rather than stress over this huge project that we have no familiarity with we decided to move our focus to a project that was a little closer to home.

Since Austin and I are limited by being the only group that has 2 teammates in it we decided to use a framework that we've been working on with Dr. Hajja as a means to bridge the knowledge gap that we were having with Amara. The project that we are working on with Dr. Hajja is a Django website that hosts dynamic teaching software for students here at the CofC! After deciding to switch our project for the third and final time we have been reaping the benefits by developing upon something that is close and familiar to us already. We understand the framework enough to know which methods need testing to improve core functionality and what possible developments we could do to assist in our further development on this project outside of CSCI362.

Chapter Two

Running the Test Suite Provided

In order to run the Amara project's test suite, the following command must be entered:

`sudo bin/dev test`

The screenshot pictured below displays the output.

```
Starting unisubs_cache_1 ...
Starting unisubs_cache_1 ... done
Starting unisubs_queue_1 ... done
nosetests --logging-clear-handlers -selenium.webdriver.remote.remote_connection --with-xunit --logging-level=WARN --xunit-file=nosete
sts.xml --where=/opt/apps/amara --verbosity=1
Creating test database for alias 'default'...
.....................................................................................................................
.....................................................................................................................
..................S..................................................................................................
.....................................................................................................................
....................................................................................................F................
......................................................................................S........S...
....................................................................
====================================================================
FAIL: test_subtitle_list (subtitles.tests.test_types.SubtitleTypesTest)
--------------------------------------------------------------------
Traceback (most recent call last):
  File "/opt/apps/amara/apps/subtitles/tests/test_types.py", line 33, in test_subtitle_list
    self.assertEqual(len(l), 13)
AssertionError: 9 != 13

--------------------------------------------------------------------
Ran 997 tests in 148.444s

FAILED (SKIP=3, failures=1)
Destroying test database for alias 'default'...
```

This output reflects an **AssertionError** (failure) on line 33 of **test_types.py** inside the **test_subtitle_list** method, seen here:

```python
class SubtitleTypesTest(TestCase):
    def setUp(self):
        pass
    def test_subtitle_list(self):
        l = SubtitleFormatListClass(ParserList, GeneratorList)
        self.assertEqual(len(l), 13)
```

The terminal shows that **l** has the value 9 when **self.assertEqual(len(l),13)** is called, which could indicate the presence of a bug within the **__init__()** function below inside the **SubtitleFormatListClass**, or it could indicate that the value 13 used for comparison is incorrect.

```python
class SubtitleFormatListClass(dict):
    def register(self, format):
        file_type = format.parser.file_type

        if isinstance(file_type, list):
            for ft in file_type:
                self[ft.lower()] = format
        else:
            self[file_type] = format

    def __getitem__(self, item):
        return super(SubtitleFormatListClass, self).__getitem__(item.lower())

    def __init__(self, parsers, generators, for_staff=False):
        super(SubtitleFormatListClass, self).__init__(self)
        for file_type in list(set(parsers.keys()).intersection(set(generators.keys()))):
            self.register(SubtitleFormat(parsers[file_type], generators[file_type], for_staff=for_staff))
    def for_staff(self):
        return [f for f, val in self.items() if val.for_staff]
SubtitleFormatList = SubtitleFormatListClass(ParserList, GeneratorList, for_staff=False)
```

Unfortunately, the code lacks documentation that explains potential sources of this inconsistency. It is unclear where the value 13 is derived, just as it is unclear how the instantiation of **l** gets its value. Analyzing this further, though, one sees that instantiation of the **SubtitleFormatListClass** takes two arguments: **ParserList** and **GeneratorList.** The following depicts the output of running the test suite with the **length of these lists printed** at the time they are passed to instantiate **SubtitleFormatListClass:**

```
Starting unisubs_cache_1 ...
Starting unisubs_db_1 ...
Starting unisubs_queue_1 ... done
nosetests --logging-clear-handlers -selenium.webdriver.remote.remote_connection
--with-xunit --logging-level=WARN --xunit-file=nosetests.xml --where=/opt/apps/a
mara --verbosity=1
Creating test database for alias 'default'...
.................................................................................
.................................................................................
.................................................................................
..........................................S.....................................
.................................................................................
.................................................................................
.................................................................................
...............................................................................
Length of ParserList =  10
Length of Generator List =  10
```

Now, referring back to the __init__ method defined in **SubtitleFormatListClass,** it can be seen that a for loop is used, iterating until reaching the **length of the intersection between the parsers.keys() set and the generators.keys() set.** In the case of **I**, the length of ParserList.keys() and GeneratorList.keys() at the time they are passed is displayed below:

```
Length of ParserList.keys() =  10
Length of GeneratorList.keys() =  10
```

Set theory implies that the intersection of these lists must be **10 or less**. To check this, a print statement was added to the test method in question (i.e. **test_subtitle_list**) to **display the length of the intersection list as well as its contents.** The result was interesting:

```
Length of parsers/generators keys intersections list:  9

List of parsers/generators keys intersections:
xml

ass

sbv

dfxp

json

vtt

srt

txt

ssa
```

The length of the intersection list, on which the for loop inside of the SubtitleFormatListClass depends, was **9**: the same value of **I** that raised the **AssertionError.** This raises the questions:

- Should the parser and generator lists have lengths of at least 13?
- Should 13 be a constant value for comparison? If so, why?

Now, while our understanding of this bug has advanced, the lack of documentation explaining the **hardcoded value 13** prevents us from understanding why this value is important, and makes it difficult to find a way to remove the bug.

Creating Our Own Test Plan

Hardware / Software Requirements:

Finding a directory that wasn't tested completely or that did not have at least some type of python testing class attached to it was nearly impossible so we considered appending to an existing one in hopes of expanding their testing in whatever way quantifiable. The given requirement for this second deliverable was to have five test cases available and thus the search for five unimplemented test methods was on. After searching through the repository for hours to locate untested methods we stumbled upon the **unisubs/apps/notifications** subdirectory. There is a testing class already implemented with it, however they didn't seem to fully test the entire notifications directory. As learned from readings past, it is near impossible to test absolutely everything within a project of this size and complexity, particularly in consideration of the **pesticide paradox**. Upon searching the notifications directory, and cross referencing with the tests.py class within it we were able to locate some methods that could be tested:

- Found in **signalhandlers.py**
    - on_team_video_save(sender, instance, created, **kwargs)
    - on_team_video_move(sender, destination_team, old_team, **kwargs)
    - on_team_member_delete(sender, instance, **kwargs)
- Found in **models.py in class TeamNotification (models.Model)**
    - is_in_progress(self)
    - set_number(self)

Additionally, since the applications in the Amara project were built using **Django,** a "high-level Python web framework"[4] with which both of us are familiar[5], we have ideas for other unit tests that are particularly related to the user interface (the Amara web page).

Things to Know

Every Django project contains (from the start) both a **urls.py** file and a **views.py** file (among others). The **urls.py** essentially provides a table of contents. When a specific URL is requested, Django searches through the list **urlpatterns** defined in that file until it finds a pattern which

---

[4] "Django: The Web framework for perfectionists with deadlines." https://www.djangoproject.com/. Accessed 3 Oct. 2017.
[5] We use Django extensively in our shared course **CSCI 399: Developing and Deploying a Dynamic Web Framework to Enhance Teaching in the Field of Computer Science** that we are taking with Dr. Hajja.

matches the requested url, at which point it calls a specified function *corresponding* to that pattern - these functions are defined in **views.py**, and they each return a predefined HTML page.

Views.py Method Testing

The first method defined in views.py is **home** - this function, invoked when the requested URL matches the pattern **url(r'^$', 'views.home', name="home"),** returns an HTML file called **unisubs/templates/oldhome.html,** the contents of which are seen below:

```
1    {% extends "hands_base.html" %}
2    {% load cache %}
3    {% block body_attrs %}id="index"{% endblock %}
4
5    {% block main_content %}
6      {% include 'hands_main.html' %}
7    {% endblock %}
```

Note that the oldhome.html file extends a much larger HTML file called **hands_base.html**, which contains all of the HTML tags needed for the creation of the Amara.org home page. This file then, in testing, plays the role of the **oracle** - the expected output of the **home** function. In order to test the function, a URL request will be made matching the regex pattern **r'^$'** (i.e. https://amara.org), the output HTML will be parsed and compared to the HTML *expected*. If there are any differences, these will be documented, as will the *number* of differences between the two files. This exact HTML-to-HTML comparison method will be used for four of the five test cases specified below (Tests 001, 003, 004, 005) in our **TestCaseSpecifications.txt** file.

- Test 001
- the test will be a single part in a dual test scenario where
we differentiate between current status of the account and cross
 check it with the expected output if the method we are testing.
- the method being tested is the logout(request)
   found in apps/auth/views.py
- the test inputs would be the current status of the account,
which is logged in and then we will use a GET URL request where the
 url pattern will match the regex (r'^logout/').
- expected output would be the account status change to logged
 out and redirection to the home page, and unsetting of the session cookie.

- Test 002
- this test will be a test to select a different destination location for a video file
- the method being tested is the
   on_team_video_move(sender, destination_team, old_team, **kwargs)
   method from unisubs/apps/notifications/signalhandlers.py
- test inputs will include the video file being reassigned
   its team tag, and the old team/ new team tags that the video will be moved to.
- expected output would be finding the video having
 acquired the new team tag as specified in the destination_team parameter.

- Test 003
- this test will be a single operation test where we scan
an HTML creation method and parse strings for accuracy on
 whether the method executed properly
- the method being tested is billing(request) in apps/teams/views.py
- test inputs would include a GET request from the URL
in order to populate the billing information for the Amara
 website. We would then cross-check
this information with the given html in the billing(request)
method within app/teams/views.py
- the outcome should be an exact html replica between the
 two being cross-checked via a string parsing and checking
 function in our testing framework.

- Test 004
- this test will be a single operation test where we scan an
 HTML creation method and parse strings for accuracy on whether
 the method executed properly
- the method being tested is user_menu_esi(request) in /views.py
- test inputs would include a GET request from the URL in
order to populate the user-menu.html doc. We will also be
cross checking the text from this html
doc similarly to the billing(request) test method above,
only difference will be in the information to check will be
acquired from /templates/future/user-menu.html.
- the outcome should be an exact html replica between the
 two being cross-checked via a string parsing and checking
 function in our testing framework.

```
├ Test 005
- the test will be a single part in a dual test scenario
where we differentiate between current status of the account
 and cross check it with the
expected output if the method we are testing. (this is the second part of Test 001)
- the method being tested is the logout(request) found in apps/auth/views.py
- the test inputs would be the current status of
 the account, which is logged out and then we will
 use a GET URL request where the url pattern
will match the regex (r'^logout/').
- expected output would be the account status change
 to logged out and redirection to the home page, and unsetting of the session cookie.
```

Amara Project Test Schedule:

10/10/2017 - get first test complete while using the script that was developed from Deliverable 2.

10/17/2017 - Edit the documentation and continue working on implementing test cases.

10/24/2017 - Make the framework necessary to test the 5 tests with the help of our script we developed earlier.

10/31/2017 - Develop the framework that will be testing our test plan and utilize 20 additional test cases.

Chapter Three

The Next Phase

During weeks leading up to Deliverable 3 we worked closely with Dr. Hajja to develop ideas for certains portions of the architecture that may need testing. Upon convening several times during the weeks we developed a greater understanding for where our testing framework would need to focus during its implementation. We determined that the view.py class needed to be the focus of our testing framework and ultimately housed many of the functions for which we could develop unit test for.

During these weeks we also tweaked our script to more closely resemble the framework that we wished to see when executing the automated testing. The script now displays much more informative user interface display messages and allows for a much more readable and pleasing testing framework rather than just straight executing our programs and displaying the results of each execution.

For our first five test cases we deduced that the checkIfOverflow(value1, value2) function would provide a necessary proof of concept for our testing development ideology. Upon developing the test cases for this function we also realized that it was feasible to potentially develop more than twenty five test cases as there are a whole host of untested functions within the views class: addTwoBinaries(binary1,binary2,length),

The Script

1. The script runAllTests2.sh that we developed follows the following path of execution:
2. cd into testCases subdirectory
3. Repeat the following for every testCase(X).txt file
    a. initialize COUNTER to 1 (to be used to assign names to the lines of the .txt file)
    b.   if current directory is not /testCases, cd into testCases
    c. Repeat the following for each line of the txt file
        i.   if COUNTER is 1 then testCaseNum is the line, print the test case number
        ii.   if COUNTER is 2 then reqToTest is the line, print reqToTest

iii.   if COUNTER is 5, then args is the line (to be passed to the function)

iv.   if COUNTER is 6, set oracle = line, cd into testCaseExecutables, and execute the testCase(testCaseNum).py file while passing in args and oracle

4. increment COUNTER

The above is an outline of the runAllTests.sh script, which is called with the terminal command:

```
"sh runAllTests2.sh"
```

Once the tests are developed by using similar implementation to that of our first five cases we plan on looking into filling a report with the returned test-made data via a text file and deliver our framework to Dr. Hajja.