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. 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 (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by the developers were functional, and we then realized that the mWater application we were working with had been superseded by a new version (found at **github.com/mWater/app-v3**).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 lack of sufficient documentation to reference), we decided to move in another, more hopeful, direction - a direction that merges Dr. Bowring's CSCI 362 with Dr. Hajja's CSCI 399.

Part Two: Testing a Dynamic Web Platform to Enhance Teaching in the Field of Computer Science

This semester, we have been working alongside Dr. Hajja in developing a website that serves to

1. Provide his CSCI 250[1] students with an interactive platform for practicing and learning the course material, and
2. Allow for a smoother, easier grading process

We are developing this website using a high-level web framework called Django that was written in Python and allows for rapid and pragmatic website design. Since we are both actively working with the development of this site, and we are familiar with its Model-View-Template architecture[2], we've decided that it would be highly beneficial to us (especially as the only two-person team) to kill two birds with one script (if you will) by

[1] Introduction to Computer Organization and Assembly Language Programming
[2] "The Model-View-Controller Design Pattern ... - The Django Book." https://djangobook.com/model-view-controller-design-pattern/. Accessed 2 Nov. 2017.

developing an automated testing framework for the site. Interestingly, the Django project currently has absolutely no tests inside of it, which means two things for us:

1. There is no testing framework which we can build upon, and
2. We get to develop a testing framework from scratch for a public project whose development we are involved in.

This shift to the CSCI 250 website represents a major milestone for team YEET; it completely removes the knowledge gap that existed when we were attempting to create a testing framework for the both *outdated* and *undocumented* mWater project. We understand the project to the extent that we know which methods 1) are central to its functionality, 2) are *simple*, and 3) need to be tested. Even more than that, this website is something that's intended to be used by students here at the college, so it also serves as a way for us to enhance and promote the education of our fellow Computer Science majors. Our plan is to merge CSCI 399 with CSCI 362 in such a way that their central concepts complement and build upon each other; creating tests for the site will push us to think more about its ongoing development, and will likely reveal the multitude of benefits that testing has to offer in the world of web development.

Running the Django Project

Django makes website development a breeze, but before it can be used, a few steps need to be taken. Note that the following commands are used with Ubuntu 16.04, and other operating systems may require different approaches.

1. The Django website recommends using the latest version of Python 3, so you need to ensure that your Python version is up-to-date. Luckily, Ubuntu 16.04 ships with both Python 3 and Python 2 pre-installed, so you just need to update and upgrade the system with:

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

```
$ sudo apt-get -y upgrade
```

2. The next step is to step up a database. This project uses MySQL, which is a popular open source database management system used for many different

applications. To install MySQL, first navigate to the MySQL download page (found at https://dev.mysql.com/downloads/repo/apt/). Click on the download link, and then right click on the "No thanks, just start my download link" and copy that link's address. Now, open the terminal and navigate to a directory to which you can write:

```
$ cd
```

```
$ cd /tmp
```

Download the file with the `curl` command:

```
$ curl -OL <paste the link you copied here>
```

Now MySQL can be installed:

```
$ sudo pkg -i mysql-apt-config
```

Refresh the apt package cache in order to make available the new software packages, and then you can delete the file you just downloaded:

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

```
$ rm mysql-apt-config*
```

Now that the repository is added and the package cache is refreshed, you can install the latest MySQL server package using the following apt-get command:

```
$ sudo apt-get install mysql-server
```

Be sure to enter 'y' when prompted to approve the installation; next, when prompted, choose a secure root password and enter it twice to complete the process. You can then check to determine whether MySQL is installed and running using systemctl:

```
$ systemctl status mysql
```

3. The next step is to install the Django web framework. This walkthrough explains how to globally install it with pip.
    a. If you don't have pip, install it:

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

```
$ sudo apt-get install python3-pip
```

    b. Now you can install Django, and subsequently verify the installation:

```
$ sudo pip3 install django
```

```
$ django-admin --version
```

4. A database needs to be created for the Django project with MySQL. To log in to an interactive session with MySQL, type:

```
$ mysql -u root -p
```

For simplicity, we can name our database after the Django project itself once in the interactive session (remember to end all commands here with a semicolon):

```
$ CREATE DATABASE csci250 CHARACTER SET UTF8;
```

Now, you can create a database user in order to be able to connect to and interact with the database you've created. Afterward, give that user access rights:

```
$ CREATE USER <yourusername>@localhost IDENTIFIED BY '<yourpassword>';
```

```
$ GRANT ALL PRIVILEGES ON csci250.* TO <yourusername>@localhost;
```

To make your changes available during the current session, flush them, and then you can exit.

```
$ FLUSH PRIVILEGES;
```

```
$ exit
```

Now install the mysqlclient package that will enable you to use the database csci250 you configured with the following command:

```
$ pip install mysqlclient
```

5. Now that a database has been created, you need to configure the Django settings file so that the project can use that database.
    a. Open the file csci250/settings.py and find the DATABASES section. Note that this indicates that the engine already points to the mysql backend. You need to ensure that the following assignments are included in this section:

        'NAME': 'csci250'

        'USER': '<yourusername>'

        'PASSWORD' : '<yourpassword>'

        'HOST' : localhost

    b. Save and close the settings.py file.
    c. Now you can "migrate" the data structures defined in the models.py file of the Django project to the database you configured and test the server. To do this, use the following commands to create and apply migrations (make sure you are in the same directory as the manage.py file before trying to enter these):

        ```
        $ python manage.py makemigrations
        ```

        ```
        $ python manage.py migrate
        ```

    d. You might also want to create an administrative account. To do this, enter:

        ```
        $ python manage.py createsuperuser
        ```

e. To find out whether the database is functioning properly, you can start up the development server with:

```
$ python manage.py runserver
```

This command should output some information telling you something like "Starting development server at http://127.0.0.1:<port number>". Opening this in a browser will lead you to the CSCI 250 website.

f. Congratulations, you've finished the walkthrough!

Chapter Two

After spending about a week studying the CSCI 250 project and looking for testable methods, we found a few that are seemingly very testable, all of which deal with basic concepts and operations covered in the course - conversions between decimal and binary representations, conversion between C and mips, and even checking for overflow in binary addition, for example. Fortunately, all of these methods are localized in one file: the views.py file of the project. For the first five test cases, we are planning to focus primarily on the checkIfOverflow method, because this method is simple enough to allow us to figure out how our script will work while also working through some test cases. As essentially stated in the method signature, this method accepts two binary numbers, adds them, and determines whether their sum causes an overflow (i.e. requires one more bit to be represented than is required by the operands).

Here is the method definition of checkIfOverflow:

It is interesting to note that, in this method, an overflow will only occur if the sum of the two values is greater than 31. This means that all inputs to this method are taken to be represented with a maximum of five bits; any output that needs more than five bits to be represented is an overflow. With that in mind, this method, as is, cannot be used to test if, say, 01 and 11 (two-bit addition) creates an overflow - which it does - because although their sum requires an additional bit, that value is only 4, which is not greater than 31.

First 5 Test Cases

Note that the format of these test cases, which are intended to be dynamically read by our **runAllTests.sh** script, is (in order by line): test case number, requirement tested, component tested, method tested, inputs, and oracle.

The following are the first five test cases we plan to use, all of which test the functionality of the checkIfOverflow method defined in views.

Test Case 1:

- 1
- check if adding two binary numbers creates overflow
  - views
  - checkIfOverflow
  - 11111,00000
  - No overflow

Test Case 2:

- 2
- check if adding two binary numbers creates overflow
  - views
  - checkIfOverflow
  - 11110,00001
  - No overflow

Test Case 3:

- 3
- check if adding two binary numbers creates overflow
  - views
  - checkIfOverflow
  - 01111,01000
  - No overflow

Test Case 4

- 4
- check if adding two binary numbers creates overflow
  - views
  - checkIfOverflow
  - 11111,11111
  - Yes (an overflow will occur)

Test Case 5

- 5
- check if adding two binary numbers creates overflow
  - views
  - checkIfOverflow
  - 11111,00001
  - Yes (an overflow will occur)

Test Plan

10/12/2017 - Find an adequate HTML table template to incorporate into the final test report, complete the basic structure of the script and get the first five test cases running. Consult with Dr. Bowring and other teams to get assistance with Python testing framework development. Consult with Dr. Hajja to determine if there are any elements of the site which need and lack testing.

10/17/2017 - Edit documentation and continue working on implementing the rest of the 25 test cases. Adjust the script as needed, respond to feedback from Dr. Bowring.

10/24/2017 - Make improvements to the script, consider adding test cases if 25 have already been completed.

10/31/2017 - Have the testing framework complete with all 25 (or more) test cases and ready for presentation to the class.

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"
```

Note that as we progress through this test framework development, we are consistently meeting with Dr. Hajja seeking advice for new tests, as well as sharing some problems which our existing framework has shed light upon. Some of the methods defined in the project work only in a very fragile sense, and are easily broken by slight input variations. These are things that we, as students helping him develop the site, can fix.

How Do You Run The Tests?

1. Clone the github repository from https://github.com/CSCI-362-02-2017/YEET.git
2. Open Terminal (Note that this project will run on UNIX or Linux!)
3. Cd into YEET/TestAutomation/scripts
4. Enter the command sh runAllTests2.sh

Note: The script is designed in a way that allows for independent creation and modification of the test case text files as well as the test case driver python files. The script does not need to be modified as test cases are added, it simply reads from the files that exist in the TestAutomation directory.

Chapter Four

Scaling Up

For the past week we have been working on scaling up our framework to incorporate an additional 20 test cases (between five and 6 cases per unique method). The feedback from our last presentation indicated that, while our test cases were formatted nicely, our framework possessed one core problem: it isolates the methods it tests from the project being tested as a whole, and this isolation is never a good thing. When testing, you always want to ensure that you're testing *live code* - that is, the most up-to-date version of the project. Isolating (copying and pasting) the method bodies into their own executable files and then executing the tests using these copies is not helpful, so we needed to find a way to let our script pull from the **views.py** file dynamically at runtime to allow for the testing framework to actually provide some meaningful results.

Naturally, our first approach was to use import statements in the executable files so that the main method in each of them could simply call the necessary method defined in views.py. However, the fact that this is a Django project created some issues associated with configuration, particularly with the settings.py file. The following is an example of one of the problems we encountered.

We, admittedly, worked for about 10 hours at the library attempting to figure this issue out, but it ultimately came down to our realization that simply importing these methods was not going to work, so we decided to move in a slightly more difficult, but more hopeful, direction.

The Fix

After meeting with Dr. Bowring about the issue, we realized that the quick and dirty way of solving this problem would be to write an additional script for pulling a given method out of the views.py file and copying it into the executable for testing - that's exactly what we did. A script called **getMethodBody.sh** was written that, when given the arguments of 1) the file name (views) and 2) the method name, navigates to that file, and uses the **sed** command to copy the given method into a temporary text file whose lines are then prepended to the executable before the definition of the main method. This script (or sub-script) is called by our higher-level **runAllTests.sh** script right before the driver (executable) file is executed, and this of course happens every time our script is run which ensures that the methods being tested are the most up-to-date methods from the project itself, *not* isolated and invariant copies.

Peripheral Changes

Along with this additional script, the output html table was corrected to be more visually appealing and less strenuous on the eyes for viewers. Also, more test cases were created; in fact, we surpassed the 25-test-case milestone in this deliverable through much trial and error with the framework's acceptance of the drivers. We ended up with a total of 30 test cases. We also added a link in the Final Report HTML file that hyperlinks to our Github repo just as a sort of convenient signature.

Chapter Five

For each of the methods our framework is responsible for testing, we injected a single fault. Most of these simply involved flipping comparison operators, while others involved more complex changes like reordering operations and changing index values. The following are brief explanations of fault injections accompanied by screenshots of the corresponding method bodies.

1. For this function, we changed line 309 from “if _decimal_value < 0” to “if _decimal_value <= 0, which causes the value of 0 to be treated as though it’s negative, causing a failed test case whenever 0 is input as the decimal value.

2. For this function, we injected a fault that's similar to the one we injected for the first one; we changed line 278 from "if _decimal_value < 0" to "if _decimal_value <= 0" which causes the value of 0 to be treated as a negative, meaning a '1' is prepended to the binary string.

```
268  def cofc_convert_decimal_to_sign_magnitude(_decimal_value, _length_of_binary):
269      decimal_value = int(_decimal_value)
270      intLength = int(_length_of_binary)
271      ans_binary = bin(int(abs(decimal_value)))
272      # Then we make it x number of bits where x is the 4th parameters in our list
273      #
274      while len(ans_binary) != (intLength + 2):
275          ans_binary = ans_binary[0:2] + '0' + ans_binary[2:]
276          # Finally, we make the MSB 1 if the number is negative, otherwise do nothing
277          #
278      if decimal_value <= 0: #was <
279          ans_binary = ans_binary[0:2] + '1' + ans_binary[3:]
280
281      return ans_binary
```

3. For this function, we swapped the order of lines 330 and 331 which causes about half of our test cases for this method to fail.

```
320  #NEW_FUNCTION
321  def cofc_ones_complement_to_decimal(_ones_complement_binary):
322      # First we check if it's positive; if it is, we simply convert to decimal
323      #
324      #_ones_complement_binary = int(_ones_complement_binary)
325      if _ones_complement_binary[2] == '0':
326          ans_decimal = int(_ones_complement_binary[2:], 2)
327      else:
328          # First we swap the 0s with 1s and vice versa
329          #
330          _ones_complement_binary = _ones_complement_binary.replace('1', '0')
331          _ones_complement_binary = _ones_complement_binary.replace('0', 'x') #order of first two assignments was swapped
332
333
334          _ones_complement_binary = _ones_complement_binary.replace('x', '1')
335
336          # Now we convert to decimal
337          #
338          ans_decimal = int(_ones_complement_binary[2:], 2) * -1  #was * -1
339
340      return ans_decimal
```

4. For this function, we flipped the comparison operator on line 250 from > to <. This is a very simple fault injection but it causes approximately half of our tests for this method to fail.

```
241 #NEW_FUNCTION
242 def cofc_convert_decimal_to_unsigned_binary_with_length(_decimal_value, _length_of_binary):
243     _decimal_value = int(_decimal_value)
244     _length_of_binary = int(_length_of_binary)
245     ans_binary = bin(_decimal_value)
246     while len(ans_binary) < (_length_of_binary + 2):
247         ans_binary = ans_binary[0:2] + '0' + ans_binary[2:]
248 
249     # Remove MSB
250     if len(ans_binary) < (_length_of_binary + 2):                    #Fault injection: was >
251         ans_binary = ans_binary[0:2] + ans_binary[3:]
252 
253     return ans_binary
```

5. For this function, we changed the index value on line 228 from 0 to 1, which causes one of the test cases to fail. When the string “x = x + 1” is entered, this particular conditional statement (line 227) is entered, and thus this is the input that causes a failed test case.

Chapter Six

Throughout the course of this project, we encountered quite a few obstacles, primarily as a result of conflicting class and work schedules, but also because of the simple fact that this was the first time that either of us had thought this extensively about software testing. Overall, we came out of this project with a much better understanding of the role that that automated testing can play - particularly in web development - as well as of the process of actually implementing that automation.

Lessons Learned

This project taught both of us a lot of valuable skills; we learned, perhaps most importantly, how to create an automated testing framework, but in order to do this we also had to teach ourselves a series of other "helper" skills. Script-writing, executing argument-accepting Python files from the terminal, identifying testable and untestable methods, and understanding bounds that need to be tested are some of these skills we developed over the course of the project - whether in response to requests from Dr. Bowring or just purely in pursuit of improving our understanding of automated testing.