Automated Testing Framework for Amara

Joshua Jettie and John Maruhn

CSCI 362 Software Engineering

Dr. Jim Bowering

College of Charleston

01 December 2016

TABLE OF CONTENTS

Introduction

This report is our Team Blue's experience while developing an automated testing framework for the open source H/FOSS project Amara. Amara is an award winning subtitling software that allows anyone to subtitle any video on Youtube. Amara is mostly used to translate Youtube videos allowing more people to access a wider variety of videos. Prior to choosing Amara there were a few key things that Team Blue needed to learn. No one on our team have ever used Ubuntu, Linux, or Virtual Box. We also had minimal experience with writing scripts. We also needed to learn Git and Github. Lastly we weren't very experience in programming from the command line. We as a team had to learn many different programs before we chose our open source project. Team Blue narrowed the candidates down from about thirty different projects down to Three. We were between Sigmah, Amara, and openMRS. There were many different factors that went in to whittling down the choices from what language it was written in to if we were generally interested in the project. Sigmah was free software developed to help international aid organizations manage the information from their projects and it was written in Java. OpenMRS was a community-developed, open-source, enterprise electronic medical record system platform that was written in java as well. Ultimately we decided on Amara because we were the most interested in their software and we were all comfortable with Python. The First Exercise we worked on as a team was to write a script called myList.[script extension]. This script was intended to list the top-level directory contents of its containing directory in an html file displayed in a browser. This was the first time we used bash as a scripting language. Finally We chose Amara as the open source project we would make a testing framework for.

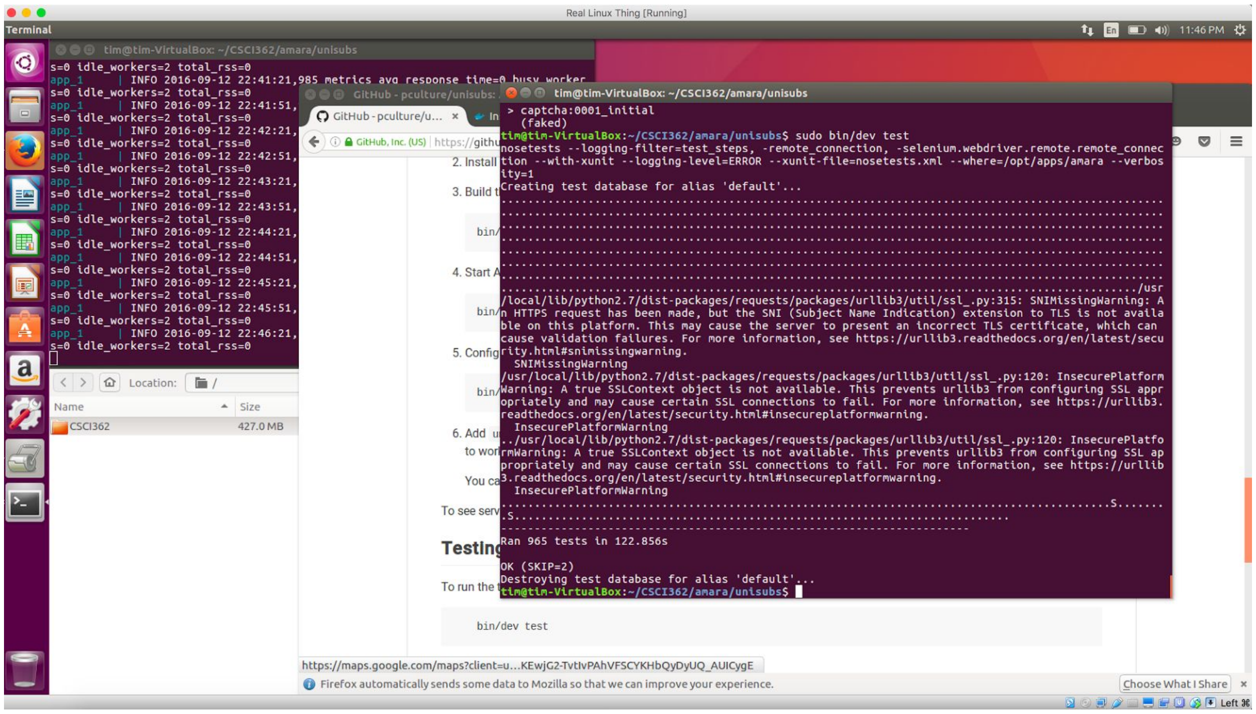
Chapter 1

DELIVERABLE #1:

Task: Checkout or clone project from its repository and build it - if you do not know how to compile the code, ask! Run existing tests and collect results. Evaluate experience and project so far.

Even before we started on Deliverable there were a couple hurdles we had to overcome. There were a few speed bumps during building Amara. First of which getting Ubuntu to work within Virtualbox. We initially set up our virtual machine with what we thought was enough memory only to find out halfway through build that we ran out of memory and would have to completely restart the build with the correct amount of memory allotted. Once we were able to get Ubuntu working within VirtualBox the next step was to install Docker within Ubuntu. With Docker working correctly with in Ubuntu we were able to clone Amara into Git and Docker. We Surprisingly did not have nearly the amount of dependency problems as the other groups. Once we had Amara running we discovered they had a build in Test suite and we were able to run the Suite. We also discovered that they had 965 tests. It seems like there may be an issue within one of the libraries, but the tests themselves are passing. Unfortunately, the test suite is kind of limited on information, which is something we may improve as we work on this project.

This is a screenshot of what it looks like to run the test suite within amara.



```
s=0 idle_workers=2 total_rss=0
app_1      | INFO 2016-09-12 22:41:21,985 metrics avg response time=0 busy worker
s=0 idle_workers=2 total_rss=0
app_1      | INFO 2016-09-12 22:41:51,
s=0 idle_workers=2 total_rss=0
app_1      | INFO 2016-09-12 22:42:21,
s=0 idle_workers=2 total_rss=0
app_1      | INFO 2016-09-12 22:42:51,
s=0 idle_workers=2 total_rss=0
app_1      | INFO 2016-09-12 22:43:21,
s=0 idle_workers=2 total_rss=0
app_1      | INFO 2016-09-12 22:43:51,
s=0 idle_workers=2 total_rss=0
app_1      | INFO 2016-09-12 22:44:21,
s=0 idle_workers=2 total_rss=0
app_1      | INFO 2016-09-12 22:44:51,
s=0 idle_workers=2 total_rss=0
app_1      | INFO 2016-09-12 22:45:21,
s=0 idle_workers=2 total_rss=0
app_1      | INFO 2016-09-12 22:45:51,
s=0 idle_workers=2 total_rss=0
app_1      | INFO 2016-09-12 22:46:21,
s=0 idle_workers=2 total_rss=0
```

```
 > captcha:0001_initial
   (faked)
tim@tim-VirtualBox:~/CSCI362/amara/unisubs$ sudo bin/dev test
nosetests --logging-filter=test_steps, -remote_connection, -selenium.webdriver.remote.remote_connec
tion --with-xunit --logging-level=ERROR --xunit-file=nosetests.xml --where=/opt/apps/amara --verbos
ity=1
Creating test database for alias 'default'...
...................................................................................................
...................................................................................................
...................................................................................................
...................................................................................................
...................................................................................................
...................................................................................................
...................................................................................................
...............................................................................................//usr
/local/lib/python2.7/dist-packages/requests/packages/urllib3/util/ssl_.py:315: SNIMissingWarning: A
n HTTPS request has been made, but the SNI (Subject Name Indication) extension to TLS is not availa
ble on this platform. This may cause the server to present an incorrect TLS certificate, which can
cause validation failures. For more information, see https://urllib3.readthedocs.org/en/latest/secu
rity.html#snimissingwarning.
  SNIMissingWarning
/usr/local/lib/python2.7/dist-packages/requests/packages/urllib3/util/ssl_.py:120: InsecurePlatform
Warning: A true SSLContext object is not available. This prevents urllib3 from configuring SSL appr
opriately and may cause certain SSL connections to fail. For more information, see https://urllib3.
readthedocs.org/en/latest/security.html#insecureplatformwarning.
  InsecurePlatformWarning
../usr/local/lib/python2.7/dist-packages/requests/packages/urllib3/util/ssl_.py:120: InsecurePlatfo
rmWarning: A true SSLContext object is not available. This prevents urllib3 from configuring SSL ap
propriately and may cause certain SSL connections to fail. For more information, see https://urllib
3.readthedocs.org/en/latest/security.html#insecureplatformwarning.
  InsecurePlatformWarning
.............................................................................................S.......
.S...........................................................................
----------------------------------------------------------------------
Ran 965 tests in 122.856s

OK (SKIP=2)
Destroying test database for alias 'default'...
tim@tim-VirtualBox:~/CSCI362/amara/unisubs$
```

Each of the dots represents a passed test, most of the text is describing an error with one of the libraries.

Chapter 2

DELIVERABLE #2:

Task: Produce a detailed **test plan**(see p. 217 text) for your project. As part of this plan, you will specify at least 5 of the eventual 25 test cases that you will develop for this software.

Amara is a web-based subtitling software that is built using Python. Because of the fact that it is web based, our team will likely run view tests, feature tests, and functionality tests in order to ensure that all of the individual parts of Amara are running correctly. We'll run these tests through a driver written in Python that will execute the tests, printing out either a "." for a passed test or an "F" for a failed test. At the end of the entire test suite, the driver will then print out the error information for all the failed tests, as well as the file location of the tests that failed.

The next step we took in order to move forward with our test plan was to figure out the requirements that we were going to be testing. We would have to isolate small testable requirements in order to be able to make a complete test suite. Below is our testing plan.

Test ID: 1

Requirement Being Tested: We are testing if the user data(first name, last name, password) matches what is stored in the database.

Component Being Tested: check_user_data(self, user, data, orig_user_data=None)

Test inputs: jettiejr@g.cofc.edu, josh, jettie

Expected outcomes: True, True, True because it is asserting true if the email, first and last name all match the provided data in the database. And we will be inputting this data into the database manually.

Path to file: unisubs/apps/api/tests/test_users.py

Test ID: 2

Requirement Being Tested: instantiates a blank user. A user with no information.

Component Being Tested: test_create_user_blank_data(self):

Test inputs: Null

Expected outcomes: 'username': 'test-user', 'email': 'test@example.com', 'first_name': '', 'last_name': '', 'full_name': '', 'bio': '', 'homepage': '' By not entering in information we are expecting that all the data to be empty strings.

Path to file: unisubs/apps/api/tests/test_users.py

Test ID: 3

Requirement Being Tested: We are testing to see if the function user_can_edit_subtitles() gives the correct output of false when we give it a user who is not staff

Component Being Tested: user_can_edit_subtitles(user, video, language_code)

Test inputs: User object who is not staff and for whom the user_can_edit_subtitles() function for workflow objects returns false, a video withe the url: https://www.youtube.com/watch?v=dQw4w9WgXcQ, en

Expected Outcomes: False, as the supplied user should not have permission to edit subtitles.

Path to File: apps/subtitles/permissions.py

Test ID: 4

Requirement Being Tested: updating the comments section when given a POST request

Component Being Tested: update_comments(request)

Test inputs: A Django request object with a POST command for a comment object

Expected Outcomes: The displayed view should contain the new comment object at the bottom of the screen.

Path to File: apps/comments/views.py

Test ID: 5

Requirement Being Tested: We are testing if we can create a video from a specific URL

Component Being Tested: test_create_videos(self):

Test inputs: "url": "http://www.example.com/sdf.mp4","langs"

Expected outcomes: True which means the video was created and ran from testhelpers.views

Path to file: unisubs/apps/testhelpers/tests.py

Testing Schedule

- Sep 27 - Choose 5 test cases to run
- Oct 18 - Create the automated testing framework
- Nov 10 - Write the remaining tests needed
- Nov 22 - Create fault injection tests
- Nov 29 – Finish writing final report

Test Recording Procedures

The driver for our testing suite will save the results of every run, including the pass/fail responses and any errors given by failing tests, to a test file, along with the date it was ran, and an ID attached to the test.

Hardware and Software

The only requirements to have Amara up and running are having Docker installed, (which for our Linux distro required Ubuntu 3.10 or later), and adding unisubs.example.com to the hosts file, pointing at 127.0.0.1

Constraints

Because of the difficulty of trying to make differing schedules mesh together, meeting with each other so that we can more effectively tackle the problem could be a large obstacle.

System Tests

The suite should not require rigorous system testing, as it is very small. Some of the tests we intend to run on the system are exceptions handling, and testing formatting when printing to an html file.

Chapter 3

DELIVERABLE #3:

Task: Design and build an automated testing framework that you will use to implement your test plan per the **Project Specifications** document.

Experience

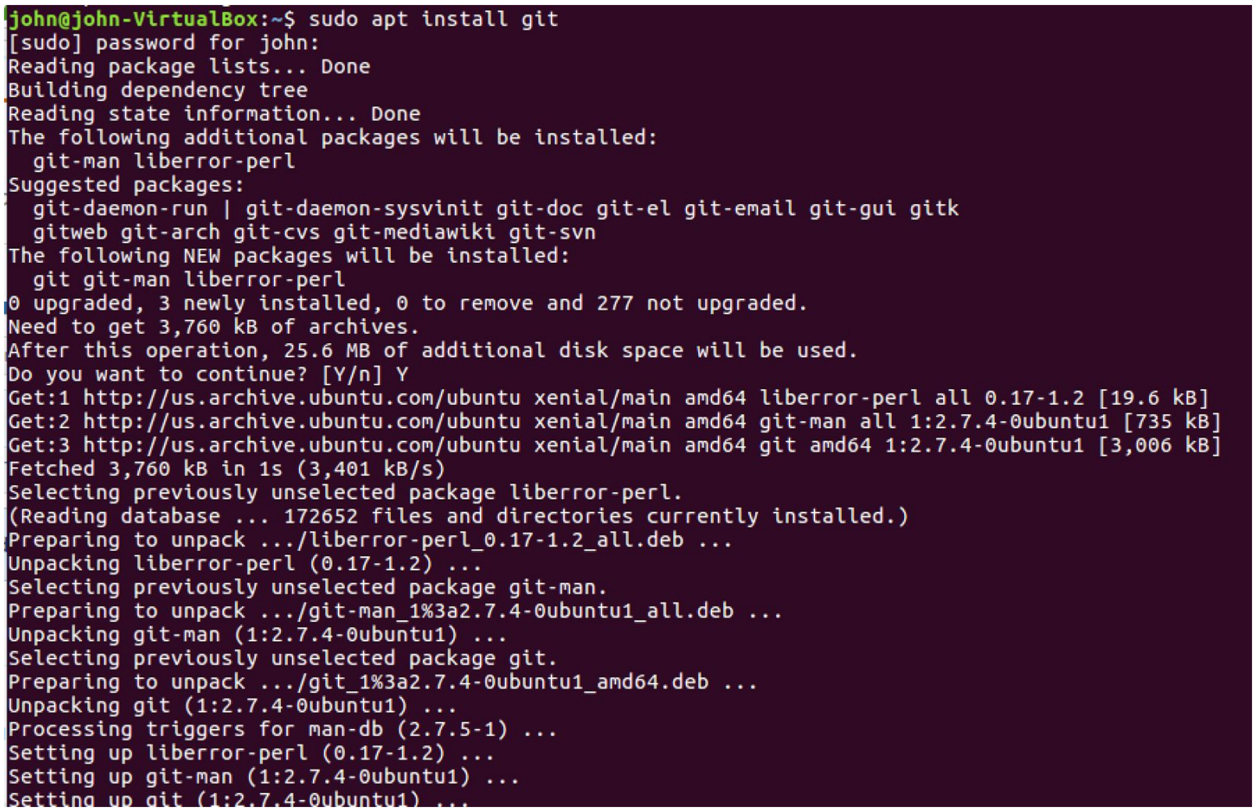
Our experience thus far has been a rough one. Our third teammate dropped the course leaving us short on manpower. We wrote our automated testing framework in Python. The goal is to make the framework know as little as possible and keep every piece of information localized to each individual test case. When we originally demonstrated our framework this was not the case. We Could not get our individual test cases to run from the commandline so we developed a driver after the fact that allow us to run individual methods from the command line. This is where we ran into many problems. An inability to easily and consistently import modules from within Amara's codebase was our primary cause of frustration because we could not access many of the classes and functions that it used. Because of this we were not able to run our original tests from our test plan because we were unable to create the necessary objects or call the appropriate functions. We were able to import from the deploy file of Amara, which contained a handful of classes and methods that we could test, but were still very limited by our inability to import modules from outside that file .

Architectural Framework

The driver of the program, *runAllTests.py*, works by fetching a list of classes located in a one of the text files in the *testCases* folder, initializing an instance of each one using Python's *eval*. Each object represents a class defined in the *scripts* folder that inherits from the abstract class *Test*. All *Test* objects define a test name for themselves that is used to look in the *testCases* folder and return all text files with filenames that include said test name. Each of these text files describes the requirement the method must meet, the component the function is located in, and the name of the function being tested. Additionally, the text files give a set of arguments and an expected result from the function's execution that the actual result is compared against. The *Test* class provides methods for retrieving that information, and automatically calls them upon initialization. For each test case file, a different entry in the *Test* object's *paramList*, *expectedList*, and *requirements* is created. The testing script then runs the function being tested with each of the parameters given in *paramList*, and compares the outcome to the corresponding entry in *expectedList*. The results are passed to an HTML formatted string that's thrown back to *runAllTests.py*, which then adds the formatted string to a larger collection string for each test. This HTML string is then written to *testOutput.html*, which is then opened at the end of the *runAllTests.py* file.

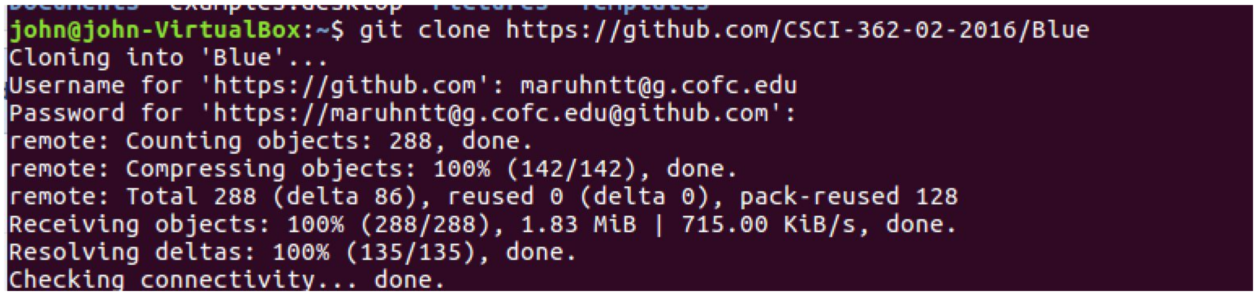
How-To:

Step 1: Install Git to your machine:



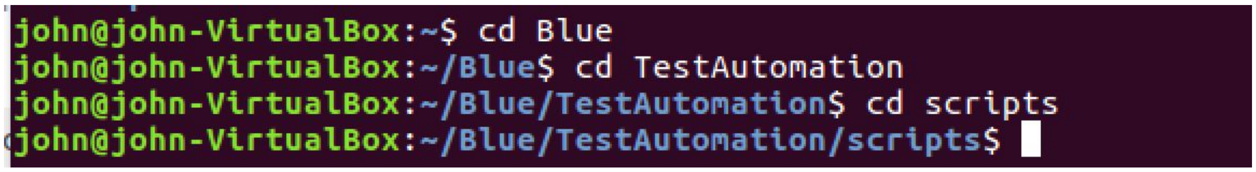
```
john@john-VirtualBox:~$ sudo apt install git
[sudo] password for john:
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  git-man liberror-perl
Suggested packages:
  git-daemon-run | git-daemon-sysvinit git-doc git-el git-email git-gui gitk
  gitweb git-arch git-cvs git-mediawiki git-svn
The following NEW packages will be installed:
  git git-man liberror-perl
0 upgraded, 3 newly installed, 0 to remove and 277 not upgraded.
Need to get 3,760 kB of archives.
After this operation, 25.6 MB of additional disk space will be used.
Do you want to continue? [Y/n] Y
Get:1 http://us.archive.ubuntu.com/ubuntu xenial/main amd64 liberror-perl all 0.17-1.2 [19.6 kB]
Get:2 http://us.archive.ubuntu.com/ubuntu xenial/main amd64 git-man all 1:2.7.4-0ubuntu1 [735 kB]
Get:3 http://us.archive.ubuntu.com/ubuntu xenial/main amd64 git amd64 1:2.7.4-0ubuntu1 [3,006 kB]
Fetched 3,760 kB in 1s (3,401 kB/s)
Selecting previously unselected package liberror-perl.
(Reading database ... 172652 files and directories currently installed.)
Preparing to unpack .../liberror-perl_0.17-1.2_all.deb ...
Unpacking liberror-perl (0.17-1.2) ...
Selecting previously unselected package git-man.
Preparing to unpack .../git-man_1%3a2.7.4-0ubuntu1_all.deb ...
Unpacking git-man (1:2.7.4-0ubuntu1) ...
Selecting previously unselected package git.
Preparing to unpack .../git_1%3a2.7.4-0ubuntu1_amd64.deb ...
Unpacking git (1:2.7.4-0ubuntu1) ...
Processing triggers for man-db (2.7.5-1) ...
Setting up liberror-perl (0.17-1.2) ...
Setting up git-man (1:2.7.4-0ubuntu1) ...
Setting up git (1:2.7.4-0ubuntu1) ...
```

Step 2: Clone our repo using the url https://github.com/CSCI-362-02-2016/Blue



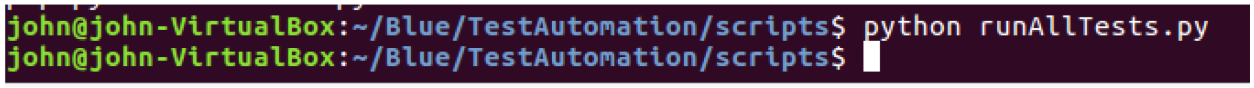
```
john@john-VirtualBox:~$ git clone https://github.com/CSCI-362-02-2016/Blue
Cloning into 'Blue'...
Username for 'https://github.com': maruhntt@g.cofc.edu
Password for 'https://maruhntt@g.cofc.edu@github.com':
remote: Counting objects: 288, done.
remote: Compressing objects: 100% (142/142), done.
remote: Total 288 (delta 86), reused 0 (delta 0), pack-reused 128
Receiving objects: 100% (288/288), 1.83 MiB | 715.00 KiB/s, done.
Resolving deltas: 100% (135/135), done.
Checking connectivity... done.
```

Step 3: Navigate to the scripts directory



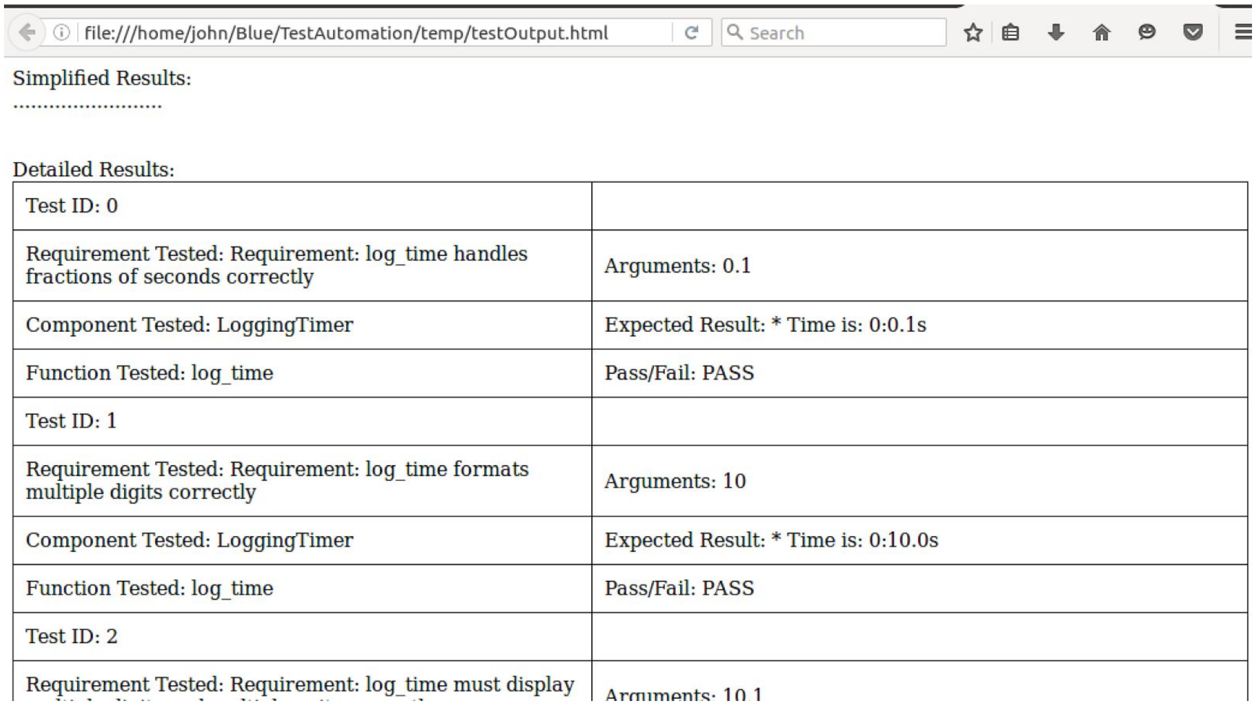
```
john@john-VirtualBox:~$ cd Blue
john@john-VirtualBox:~/Blue$ cd TestAutomation
john@john-VirtualBox:~/Blue/TestAutomation$ cd scripts
john@john-VirtualBox:~/Blue/TestAutomation/scripts$
```

Step 4: Run the file runAllTests.py with Python (will take a bit due to sleep functions, >25 sec)



```
john@john-VirtualBox:~/Blue/TestAutomation/scripts$ python runAllTests.py
john@john-VirtualBox:~/Blue/TestAutomation/scripts$
```

Step 5: An HTML page of the test results should automatically open at the end



file:///home/john/Blue/TestAutomation/temp/testOutput.html

Simplified Results:
....

Detailed Results:

Test ID: 0	
Requirement Tested: Requirement: log_time handles fractions of seconds correctly	Arguments: 0.1
Component Tested: LoggingTimer	Expected Result: * Time is: 0:0.1s
Function Tested: log_time	Pass/Fail: PASS
Test ID: 1	
Requirement Tested: Requirement: log_time formats multiple digits correctly	Arguments: 10
Component Tested: LoggingTimer	Expected Result: * Time is: 0:10.0s
Function Tested: log_time	Pass/Fail: PASS
Test ID: 2	
Requirement Tested: Requirement: log_time must display	Arguments: 10.1

Chapter 4

<u>Deliverable #4</u>

<u>Task:</u>

To complete the design and implementation of your testing framework as specified in Deliverable #3. You are now ready to test your project in earnest. You will create 25 test cases that your framework will automatically use to test your H/FOSS project.

<u>Process:</u>

We have been able to write 25 test cases that don't involve us using a false database or Django. We have been very limited in methods we can use because they all have database dependencies. The first method we found we can test was the LoggingTimer because it was one of the only classes in the entire repository that does not have database dependencies. We created 10 different test cases for this method testing it against a sleep timer to make sure the method produced the correct results. We also tested this method to make sure that we could induce a fault and we could break this method. The next method we found that had no database dependencies was the log(msg, *args, **kwargs) method what created a log of the time. Again we were very limited in the methods we could test because we were not able to get all the background dependencies to run. We tested to make sure this method was producing the correct results we also were able to make this method fail our tests by forcing it to produce incorrect results. Additionally, we tested log and log_nostar with multiple arguments in *args, in order to ensure that it could handle multi-argument calls. We then tested the cd_to_root_project function in order to see if it retrieves the correct root directory. We did this by seeing if the path generated in the function matched the directory for the function caller, which produced the correct results. Finally, we checked out if the get_docker_hosts function returned the correct error message, as we could not set up the database. It informed us that the DOCKER_HOSTS ENV variable was not set, which was the expected output. The last method that we could test that did not require a database or Django dependencies was the log_nostar(msg, *args, **kwargs). This method simply took the log methods output and stripped the star from it in order to make the results from the log method look more uniform.

Results:

After analyzing our results we were able to determine that our test cases were working as intended. Some of the issues we ran into were when we wanted to produce an incorrect result it would cause the method to crash. Our results showed that our driver was not able to handle exceptions very well.

Problems:

Amara has proved to be nothing but problems because of dependencies. When we first started writing our test cases we planned on testing items against a false database. This proved to be very complicated because we could not get their database to work because both of us had no experience with Docker. Docker was just the first problem. Once we got Docker running we ran into the problem that would ultimately make it so we couldn't use the Database and that was

Django. Both of us spent many hours researching how to make Django work. We were able to get Django installed but could never use it to build our false database. Because of this we had to dig through the Amara repository and find methods that did not use a Django dependency which turned out to be pretty much all of them. Another problem we kept running into was python pathing and how to run methods from the command line with our driver.

Chapter 5

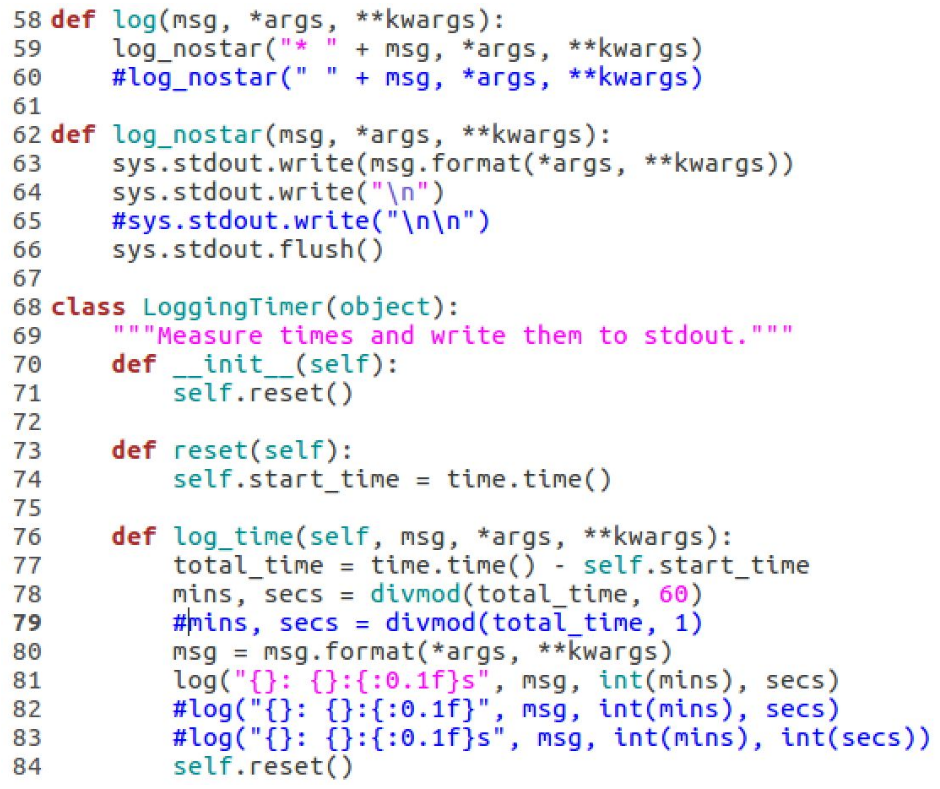
Deliverable #5:

Task: Design and inject 5 faults into the code you are testing that will cause at least 5 tests to fail, but hopefully not all tests to fail. Exercise your framework and analyze the results.

Our Experience

Our experience with fault injection proved to be easier than expected compared to the last deliverables. I think it is because it is a lot easier to make code not work than to make code work. We began with a very small bank of methods so it was really just figuring out tiny bugs we could insert that would not make all 25 tests fail. We were able to inject minor faults such as changing formatting or changing the division of the seconds in order to produce the incorrect amount of seconds on a log timer.

Code prior to Fault Injections:

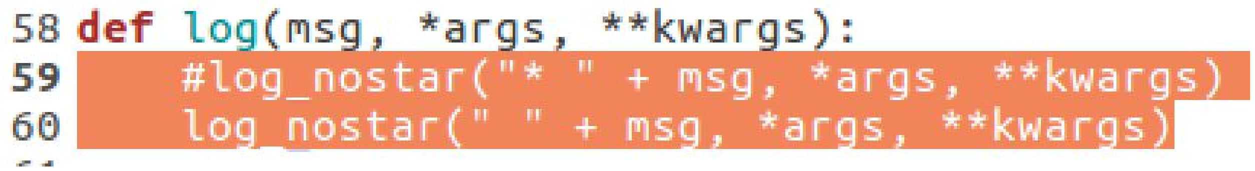


```
def log(msg, *args, **kwargs):
    log_nostar("* " + msg, *args, **kwargs)
    #log_nostar(" " + msg, *args, **kwargs)

def log_nostar(msg, *args, **kwargs):
    sys.stdout.write(msg.format(*args, **kwargs))
    sys.stdout.write("\n")
    #sys.stdout.write("\n\n")
    sys.stdout.flush()

class LoggingTimer(object):
    """Measure times and write them to stdout."""
    def __init__(self):
        self.reset()

    def reset(self):
        self.start_time = time.time()

    def log_time(self, msg, *args, **kwargs):
        total_time = time.time() - self.start_time
        mins, secs = divmod(total_time, 60)
        #mins, secs = divmod(total_time, 1)
        msg = msg.format(*args, **kwargs)
        log("{}: {}:{:0.1f}s", msg, int(mins), secs)
        #log("{}: {}:{:0.1f}", msg, int(mins), secs)
        #log("{}: {}:{:0.1f}s", msg, int(mins), int(secs))
        self.reset()
```

Fault Injection #1

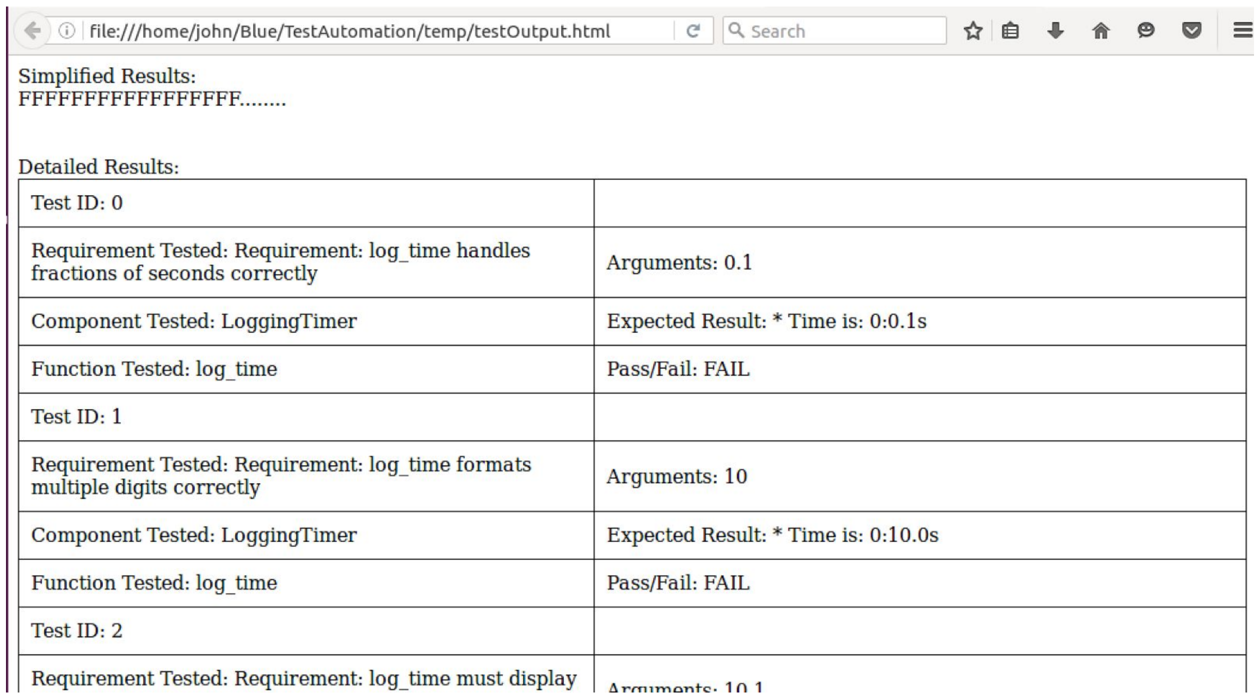
Method being tested: log(msg, args, kwargs)

The first fault we injected we in the Log method and we deleted a * with in the . The log method before the injection prints a message beginning with a * to the stdout.



```
def log(msg, *args, **kwargs):
    #log_nostar("* " + msg, *args, **kwargs)
    log_nostar(" " + msg, *args, **kwargs)
```

Results: Previous to the fault being injected all 25 tests pass. After the fault was injected 16 tests failed because because the methods required a * to be present.



file:///home/john/Blue/TestAutomation/temp/testOutput.html

Simplified Results:
FFFFFFFFFFFFFFFF.........

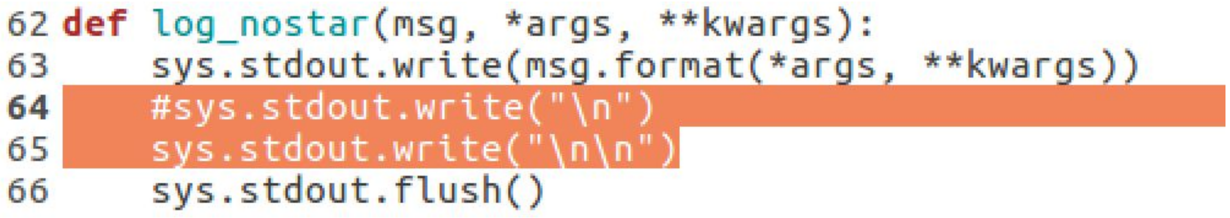
Detailed Results:

Test ID: 0	
Requirement Tested: Requirement: log_time handles fractions of seconds correctly	Arguments: 0.1
Component Tested: LoggingTimer	Expected Result: * Time is: 0:0.1s
Function Tested: log_time	Pass/Fail: FAIL
Test ID: 1	
Requirement Tested: Requirement: log_time formats multiple digits correctly	Arguments: 10
Component Tested: LoggingTimer	Expected Result: * Time is: 0:10.0s
Function Tested: log_time	Pass/Fail: FAIL
Test ID: 2	
Requirement Tested: Requirement: log_time must display	Arguments: 10.1

Fault Injection #2

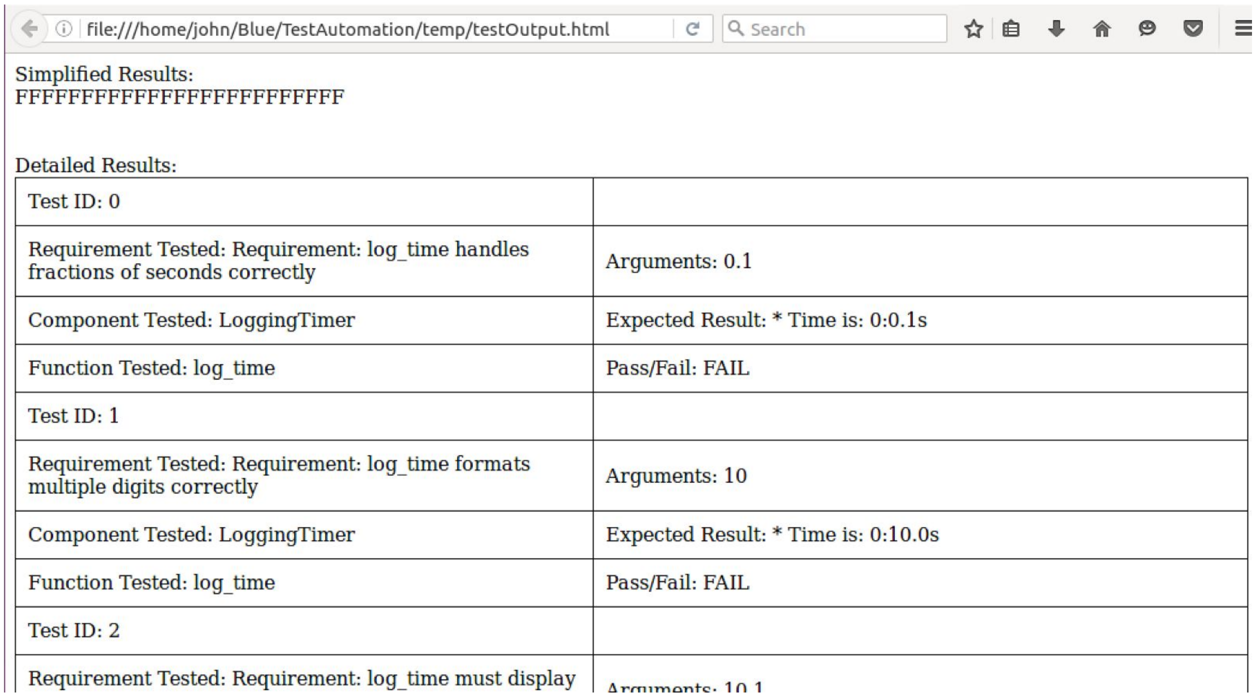
Method being tested: log_nostar(msg, args, kwargs)

The second fault we induced was to add an extra line to the stdout in the log_nostar method. This should cause all of our tests to fail because all of our tests rely on log_nostar to print to stdout with no star.



```
def log_nostar(msg, *args, **kwargs):
    sys.stdout.write(msg.format(*args, **kwargs))
    #sys.stdout.write("\n")
    sys.stdout.write("\n\n")
    sys.stdout.flush()
```

Results: Before the fault was injected all 25 tests pass and after the fault was injected all 25 tests failed because all the methods being tested used log_nostar in some capacity.



file:///home/john/Blue/TestAutomation/temp/testOutput.html

Simplified Results:
FFFFFFFFFFFFFFFFFFFFFFFFF

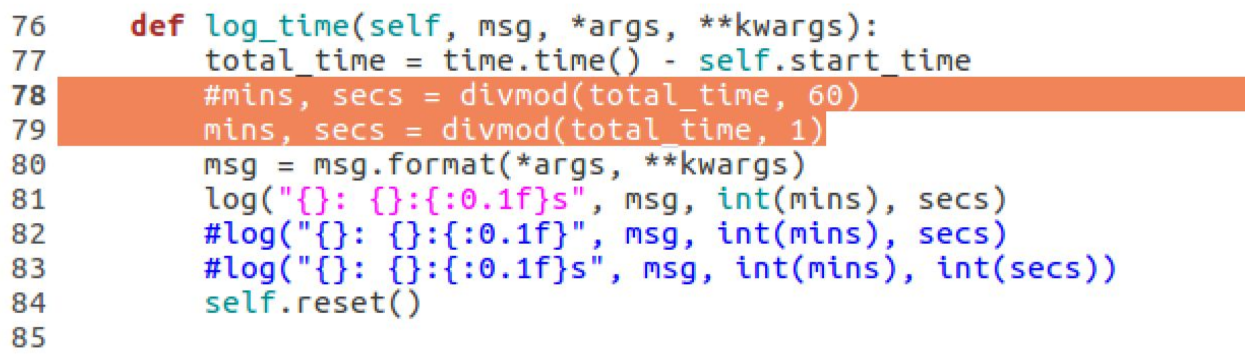
Detailed Results:

Test ID: 0	
Requirement Tested: Requirement: log_time handles fractions of seconds correctly	Arguments: 0.1
Component Tested: LoggingTimer	Expected Result: * Time is: 0:0.1s
Function Tested: log_time	Pass/Fail: FAIL
Test ID: 1	
Requirement Tested: Requirement: log_time formats multiple digits correctly	Arguments: 10
Component Tested: LoggingTimer	Expected Result: * Time is: 0:10.0s
Function Tested: log_time	Pass/Fail: FAIL
Test ID: 2	
Requirement Tested: Requirement: log_time must display	Arguments: 10.1

Fault Injection #3

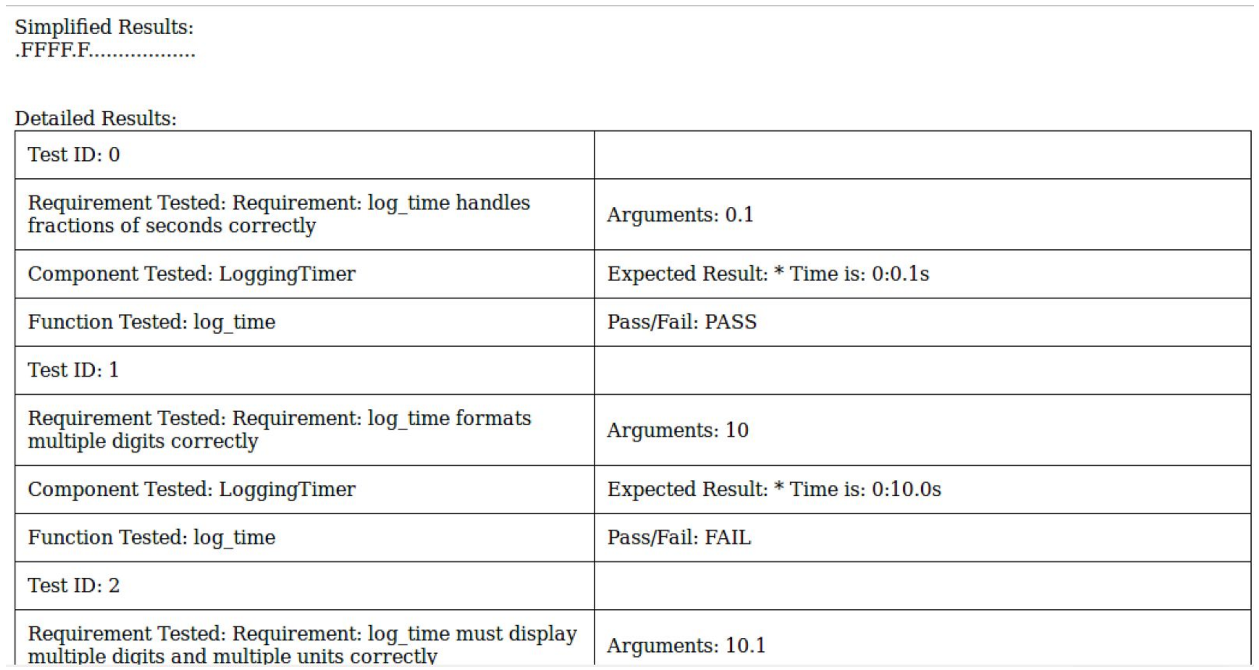
Method being tested: log_time(self, msg, args, kwargs)

This method previous to the fault injection keeps track of the time and prints the time to the stdout. The method in order to keep track of seconds in a minute divides by 60. Our fault will cause the method to divide by 1 in order to give us the incorrect amount of seconds.



```
    def log_time(self, msg, *args, **kwargs):
        total_time = time.time() - self.start_time
        #mins, secs = divmod(total_time, 60)
        mins, secs = divmod(total_time, 1)
        msg = msg.format(*args, **kwargs)
        log("{}: {}:{:0.1f}s", msg, int(mins), secs)
        #log("{}: {}:{:0.1f}", msg, int(mins), secs)
        #log("{}: {}:{:0.1f}s", msg, int(mins), int(secs))
        self.reset()

```

Results: Before the fault was injected all 25 tests passed. After we induced the fault 5 out of 25 tests failed because the seconds on the log timer were incorrect.



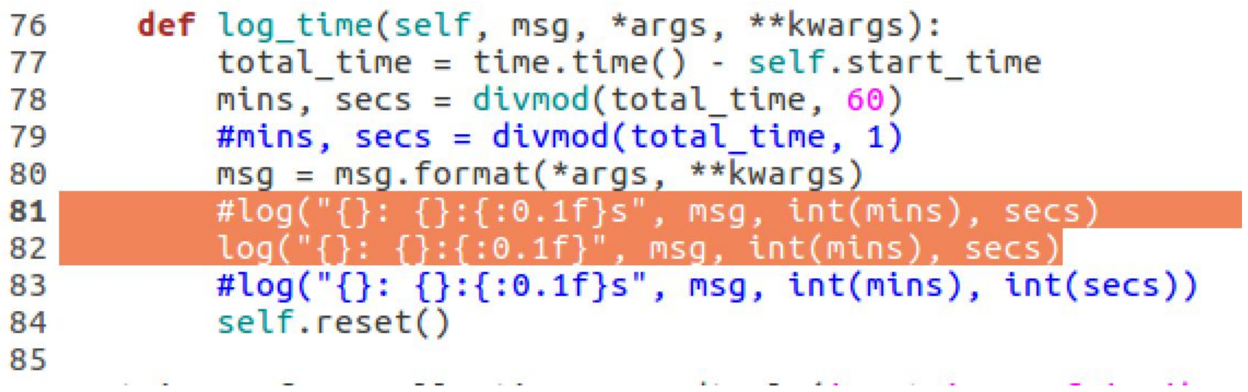
Simplified Results:
.FFFF.F..................

Detailed Results:

Test ID: 0	
Requirement Tested: Requirement: log_time handles fractions of seconds correctly	Arguments: 0.1
Component Tested: LoggingTimer	Expected Result: * Time is: 0:0.1s
Function Tested: log_time	Pass/Fail: PASS
Test ID: 1	
Requirement Tested: Requirement: log_time formats multiple digits correctly	Arguments: 10
Component Tested: LoggingTimer	Expected Result: * Time is: 0:10.0s
Function Tested: log_time	Pass/Fail: FAIL
Test ID: 2	
Requirement Tested: Requirement: log_time must display multiple digits and multiple units correctly	Arguments: 10.1

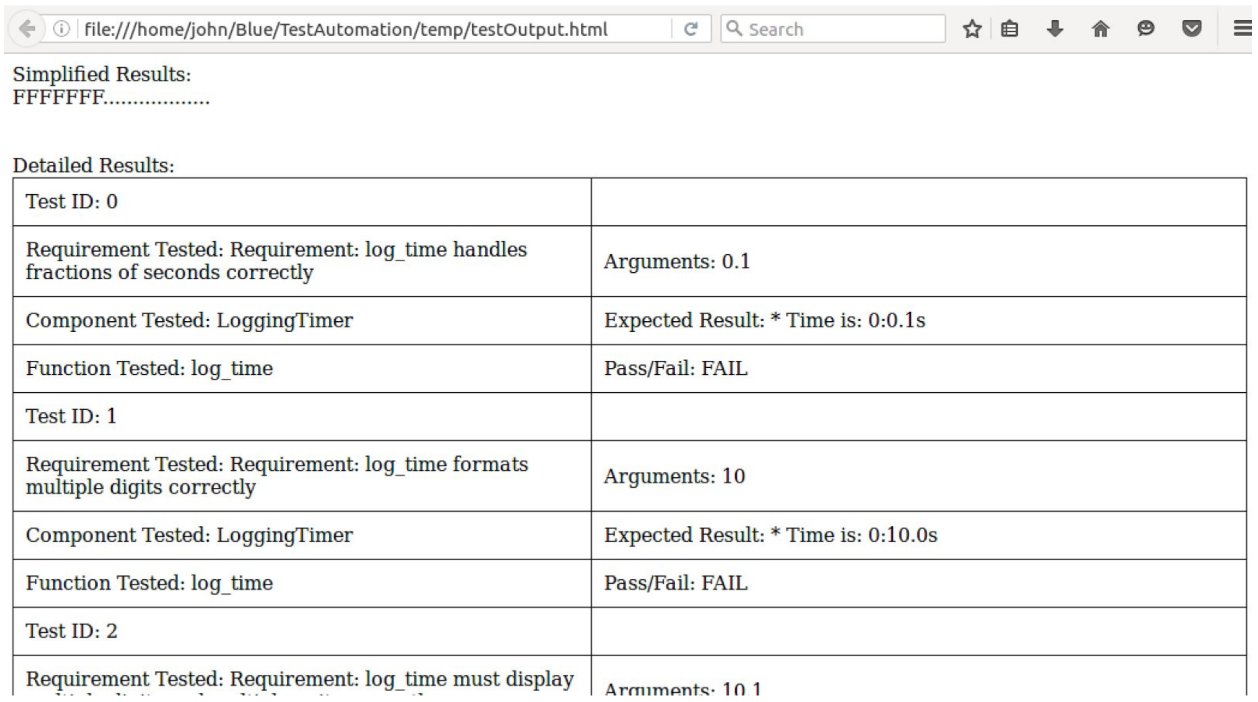
Fault Injection #4

Method being tested: log_time(self, msg, args, kwargs) Our fault we will be injecting is altering the format in which the log timer prints to stdout. Before the fault the format looks like this. Our fault will remove the “s” at the end of the stdout message causing some tests that require the correct format to fail.



```
    def log_time(self, msg, *args, **kwargs):
        total_time = time.time() - self.start_time
        mins, secs = divmod(total_time, 60)
        #mins, secs = divmod(total_time, 1)
        msg = msg.format(*args, **kwargs)
        #log("{}: {}:{:0.1f}s", msg, int(mins), secs)
        log("{}: {}:{:0.1f}", msg, int(mins), secs)
        #log("{}: {}:{:0.1f}s", msg, int(mins), int(secs))
        self.reset()

```

Results: Previous to our fault all 25 tests passed. After we induced the fault 7 out of 25 tests failed because they required an “s” indicating seconds at the end of the stdout message.



Simplified Results:
FFFFFFF..................

Detailed Results:

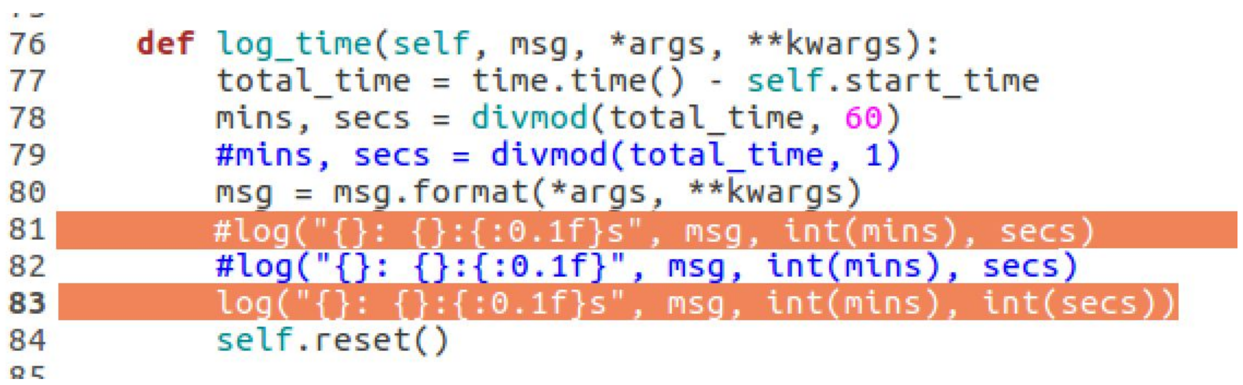
Test ID: 0	
Requirement Tested: Requirement: log_time handles fractions of seconds correctly	Arguments: 0.1
Component Tested: LoggingTimer	Expected Result: * Time is: 0:0.1s
Function Tested: log_time	Pass/Fail: FAIL
Test ID: 1	
Requirement Tested: Requirement: log_time formats multiple digits correctly	Arguments: 10
Component Tested: LoggingTimer	Expected Result: * Time is: 0:10.0s
Function Tested: log_time	Pass/Fail: FAIL
Test ID: 2	
Requirement Tested: Requirement: log_time must display	Arguments: 10.1

Fault Injection #5

Method being tested: log_time(self, msg, args, kwargs)

Previous to the fault log_time logs logs seconds as floats. The fault we will be injecting will cause the log_time method to log seconds as ints thus causing any tests that require]ing fractions of seconds to fail.

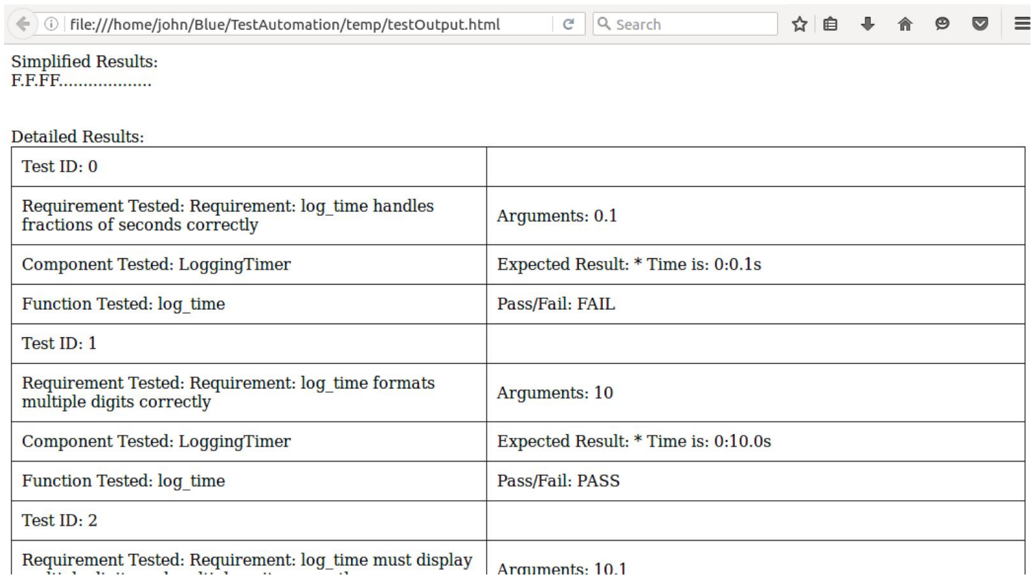
Results: Before our fault 25 tests passed and after we induced the fault 4 out of 25 tests failed. The reason they failed was that we rounded all the decimals causing any of our tests that required fractions of seconds to fail.



```
    def log_time(self, msg, *args, **kwargs):
        total_time = time.time() - self.start_time
        mins, secs = divmod(total_time, 60)
        #mins, secs = divmod(total_time, 1)
        msg = msg.format(*args, **kwargs)
        #log("{}: {}:{:0.1f}s", msg, int(mins), secs)
        #log("{}: {}:{:0.1f}", msg, int(mins), secs)
        log("{}: {}:{:0.1f}s", msg, int(mins), int(secs))
        self.reset()
```

Results:

We were able to successfully inject five different faults in order to simulate a real life scenario in which we run our tests frequently and another engineer were to come along and change our code and our tests began to fail. This demonstrates the necessity for running tests constantly because it allows us to catch any minor bugs that may have been injected by other engineers.



file:///home/john/Blue/TestAutomation/temp/testOutput.html

Simplified Results:
F.F.FF...................

Detailed Results:

Test ID: 0	
Requirement Tested: Requirement: log_time handles fractions of seconds correctly	Arguments: 0.1
Component Tested: LoggingTimer	Expected Result: * Time is: 0:0.1s
Function Tested: log_time	Pass/Fail: FAIL
Test ID: 1	
Requirement Tested: Requirement: log_time formats multiple digits correctly	Arguments: 10
Component Tested: LoggingTimer	Expected Result: * Time is: 0:10.0s
Function Tested: log_time	Pass/Fail: PASS
Test ID: 2	
Requirement Tested: Requirement: log_time must display	Arguments: 10.1