

Team 7-11

Dr. Bowring

CSCI 362-02

Fall 2020

Team Term Project

Automated UI Testing Framework

for **Wheelmap-frontend**

Justin Garrison - Janneke Morin - Matt Walter

Table of Contents

Table of Contents	2
Introduction	4
Project specifications	4
Chapter 1	6
Our project choice	6
Compiling the project	7
Ideas about testing	10
Evaluation of experience	11
Chapter 2	13
Introduction	13
The testing process/tested items	14
Requirements traceability	14
Testing schedule	15
Test recording procedures	15
Hardware and software requirements	16
Short-list of software requirements to build the project	17
Procedure to set up software requirements for testing the project	17
Constraints	18
System tests	18
Chapter 3	21

	3
Introduction	21
Framework overview	22
The Parser (Parser.py)	22
The driver (testmap.py)	23
The report	25
Test cases	26
How-to guide	28
Chapter 4	30
Introduction	30
Updates to report	30
Reflection	35
Chapter 5	36
Introduction	36
Changes and associated test cases	37
Explanation of Changes	38
Experience Report	39
Chapter 6	40
Team evaluation	40
Lessons learned	40
Project weaknesses	41

Introduction

CSCI 362, Software Engineering, is a course within the Computer Science department at the College of Charleston. It is required for Computer Science majors. The course focuses on the development process and overall practice of software engineering. In Dr. Bowring's version of the course, students experience the discipline first-hand through a team term project. In this project, they choose an open-source software project and develop an automated testing framework for it over the course of the semester.

We (Justin, Janneke, and Matt) were paired together for this project in the Fall 2020 semester. This report details our journey to creating an automated testing framework for Wheelmap. Chapters 1-5 were written over the course of the semester. Chapter 6 was completed as a reflection on the project. Our team [Github repository](#) and Wiki contain additional documentation and instructions for our project.

Project specifications

Design and build an automated testing framework that you will use to implement your test plan for your chosen software project from Deliverable #2: Produce a detailed test plan.

The testing framework will run on Ubuntu (Linux/Unix) as a script executed from the terminal. 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. In other words, each test case specification file thus contains the meta-data that your framework needs to set up and execute the test case and to collect the results of the test case execution.

The above was extracted from Dr. Bowring's Team Term Project Specification Document. To view the full testing framework specifications and example template, please follow this [link](#).

Chapter 1

Building the project

Deliverable 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.

Introduction

Prior to this first deliverable, we chose our top three candidates from a list of H/FOSS (Humanitarian Free and Open Source Software) applications that could possibly serve as student projects¹. After attempting to compile these three projects, we zeroed in on our top candidate, **wheelmap-frontend**².

Wheelmap is a crowdsourced online map to search, find, and mark wheelchair-accessible places. Wheelmap-frontend is the newer, maintained

¹ http://foss2serve.org/index.php/HFOSS_Projects

² <https://github.com/sozialhelden/wheelmap-frontend>

repository containing the Node.js/React.js-based frontend for the Wheelmap app.

Compiling the project

Fortunately, we faced relatively simple issues while compiling this project for the first time. The following table compares the repository's instructions for compiling wheelmap-frontend against our actual sequence of terminal commands:

Build Instructions	Actual Build Sequence
<pre># Environment variables cp .env.example .env # npm dependencies npm install # install transifex i18n / localization tool pip install transifex-client</pre>	<pre># clone the repository git clone https://github.com/sozial helden/wheelmap-frontend # enter the cloned repository cd wheelmap-frontend # environment variables cp .env.example .env sudo apt install npm sudo apt install transifex</pre>

<pre># start a local test web server npm run dev</pre>	<pre>npm run dev npm rebuild npm run dev</pre>
---	--

Installing Node Package Manager (npm) proved one of our biggest pain-points. This is the default package manager for the JavaScript environment node.js. After running `npm install` per the directions, we repeatedly ran into the error: `"npm WARN Local package.json exists, but node_modules missing, did you mean to in install?"`.

Running `npm rebuild` resolved this issue. After reading the documentation for npm³, we found that rerunning `npm install` or running `npm rebuild` will compare the existing "node_modules" installed by npm to those in the most current version of npm, only installing what is missing. We are still looking into why this function would resolve the issue immediately after we installed npm and, presumably, the most up-to-date packages.

Once the npm issue had been resolved, we received the message: `"Ready on https://localhost:3000"` and were able to open a local version of Wheelmap on Firefox!

³ <https://docs.npmjs.com/cli/install#algorithm>


```

janneke@janneke-VirtualBox: ~/wheelmap-frontend
[nodemon] watching: /home/janneke/wheelmap-frontend/src/server/**/*
[nodemon] starting 'node --icu-data-dir=node_modules/full-icu --inspect src/server/server.js'
Debugger listening on ws://127.0.0.1:9229/e1677a92-0774-4d7f-ada2-0361c604a3bc
For help, see: https://nodejs.org/en/docs/inspector
Using environment variables from .env file, overridden by system-provided environment variables.
Node version: v10.19.0
Warning: Built-in CSS support is being disabled due to custom CSS configuration being detected.
See here for more info: https://err.sh/next.js/built-in-css-disabled

> Using "webpackDevMiddleware" config function defined in next.config.js.
> Using external babel configuration
> Location: "/home/janneke/wheelmap-frontend/.babelrc"
event - compiled successfully
wait - compiling...
Attention: Next.js now collects completely anonymous telemetry regarding usage.
This information is used to shape Next.js' roadmap and prioritize features.
You can learn more, including how to opt-out if you'd not like to participate in this anonymous pr
ogram, by visiting the following URL:
https://nextjs.org/telemetry

[HPM] Proxy created: / -> http://classic.wheelmap.org
[HPM] Proxy created: / -> http://classic.wheelmap.org
> Ready on http://localhost:3000
event - compiled successfully

```

Figure 1: our terminal after successful compilation of the project

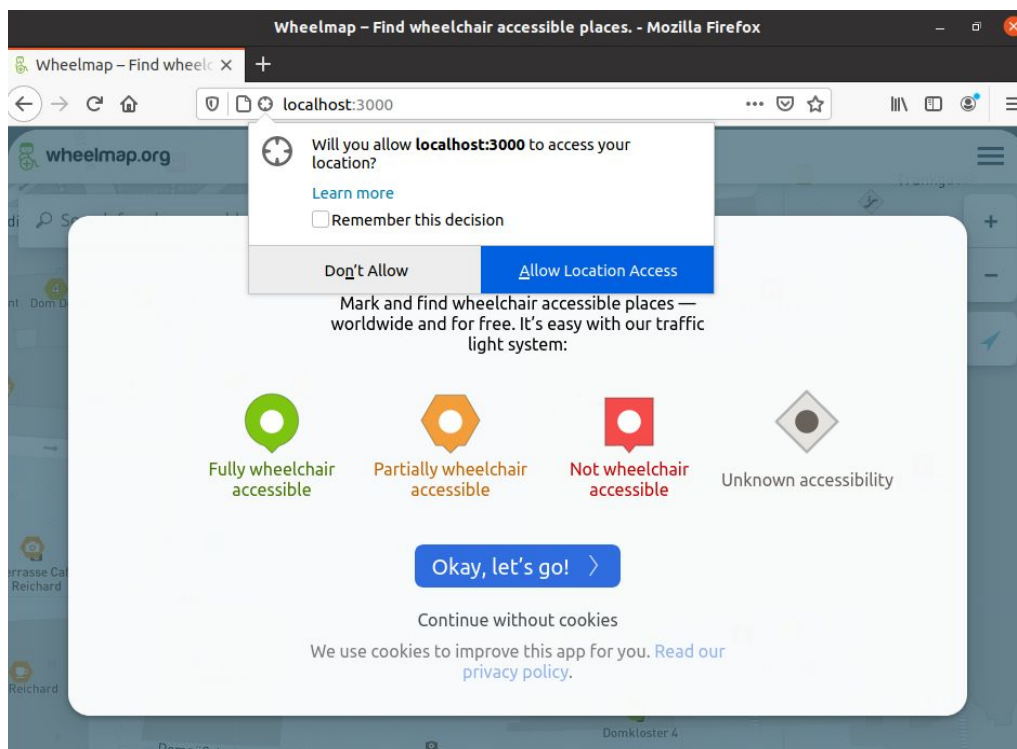


Figure 2: Wheelmap running locally on our virtual machine

Ideas about testing

The wheelmap-frontend repository README.md page has the following note about testing: “For testing the apps, we use BrowserStack - it can run test suites on various browsers and live devices. Currently, our testing happens mostly manually on BrowserStack Live, but pull requests will soon get automatic CI checks using BrowserStack Automate and BrowserStack App Automate. We thank the BrowserStack team for their great products and their support!”

We made free accounts on BrowserStack to investigate and found that it provides live testing (limited in the free version) which allows you to select a device/operating system/browser, then test a series of steps. You can also automate this testing through BrowserStack Automated. However, since we are not affiliated with Wheelmap and do not have the paid version, we were unable to access any tests they have automated and organized through the application.

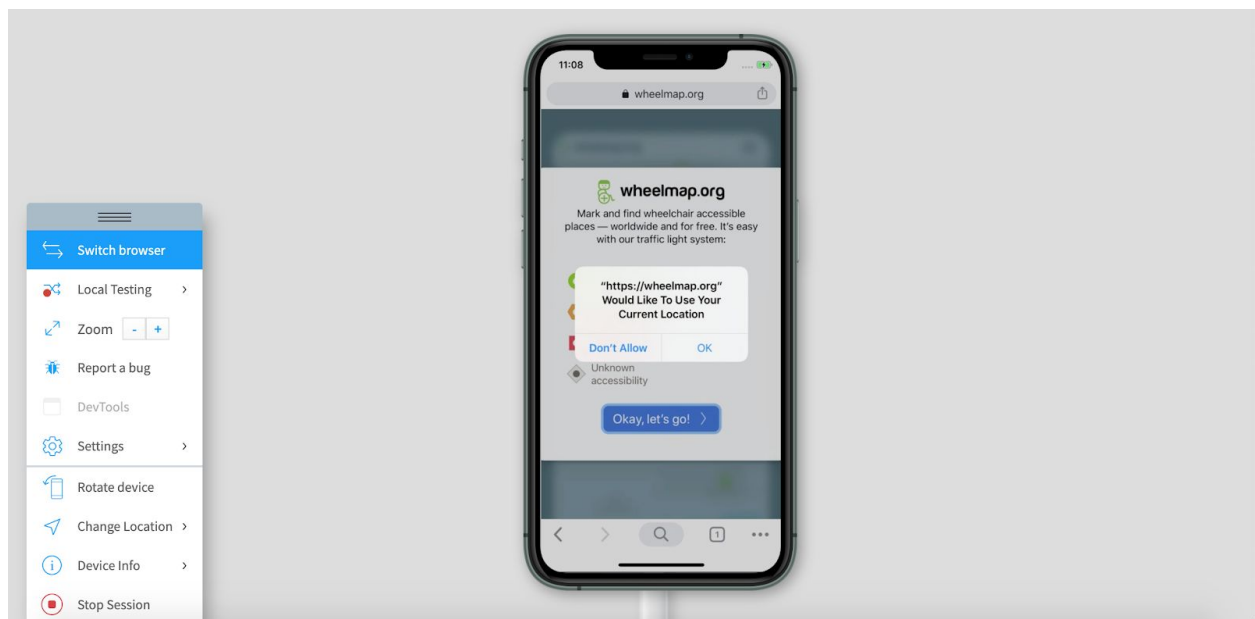


Figure 3: BrowserStack mobile testing

Within the “docs” folder of the repository, we found a testing document⁴ which outlines an extensive series of steps to test the main functionality of the wheelmap React.js frontend application. These are performed by manually clicking around the application. The document includes 28 sections with test actions like the sample ones below:

- 'welcome' dialog shown on first start
- 'welcome' dialog can be dismissed with the 'lets go' button
- tapping any button or the blurred does not trigger any reaction on other underlying components
- 'welcome' dialog can be re-opened by tapping the logo in the main menu
- Browser(tab) title matches application name

Behind the scenes, the Wheelmap team likely has these tests scripted and automated within the BrowserStack app, though it seems they have not publicized any test code. We are working on reaching out to the Wheelmap test to see if it's possible to earn access to the testing scripts.

Evaluation of experience

So far, this project has proved a great learning experience for each of us. We had the opportunity to interact with several different projects, evaluating their strengths, weaknesses, and deal-breakers before finally choosing Wheelmap. This process shed light on the importance of thorough documentation because in all cases, that was the deal-breaker for us. We ran into errors attempting to

4

<https://github.com/sozialhelden/wheelmap-frontend/blob/master/docs/TESTING.md>

compile Cadasta and Martus that were not worth pushing through when our other candidate, Wheelmap, was so clearly documented. Our next steps include learning more about the existing testing process for Wheelmap looking ahead to the second deliverable.

Chapter 2

Formulating a Test Plan

Deliverable 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.*

Introduction

In our exploration in Chapter 1, we learned that Wheelmap utilizes a paid application called BrowserStack to test the JavaScript front-end of the application. It allows for both automated and manual testing of the user interface, checking whether a given sequence of actions by the user produces the expected results. For instance, if a user clicks the filter for “Partially wheelchair accessible,” does the map view change to only mark these locations? Since BrowserStack is a paid application, we weren’t able to view the back-end code driving the tests.

As we began to think about building our own testing framework, Dr. Bowring and Matt suggested the idea of working with Selenium. Selenium is a web-based automation tool for connecting and sending commands to a web browser through Python. After experimenting with it, we have decided that Selenium is a

great fit that will allow us to replicate the kinds of tests the Wheelmap team performs through BrowserStack. Our tests will send a set of commands to the browser to navigate around the locally built version of Wheelmap, then report whether they result in the oracle. We list five of the eventual twenty-five test cases below in the **System tests** section.

The testing process/tested items

To devise the major phases of our test plan, we walked through sequences of actions a user might take within the user interface. This brought the following major categories of tests:

1. Opening the application
 - a. Asking user for location access
2. Navigation
 - a. Zooming in and out
3. Filter functionality
 - a. Filtering locations
 - b. Updating the user interface
4. Individual location options
 - a. Adding images
 - b. Opening on separate map applications
 - c. Sharing
 - d. Reporting a problem

Requirements traceability

The following requirements are tested through our first five test cases:

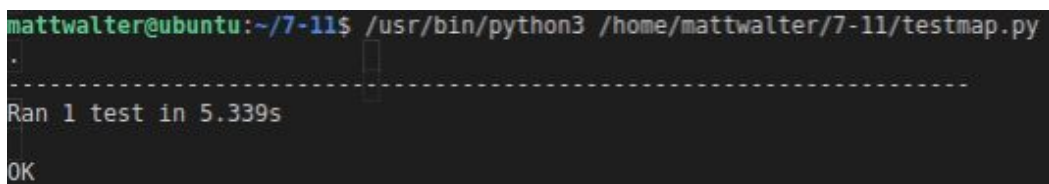
1. First-time users should be prompted with a request for location access so that the app can provide user-specific location functionality.
2. When a user clicks to enable the 'Locate Me' button, the app should respond by highlighting the location icon in blue, indicating the mapview has shifted to the scope of the user's location.
3. When a user clicks to disable the 'Locate Me' button AND the scope of the map view is already focused on the user's location, the app should respond by un-highlighting the location icon.
4. When a user hovers over a button - for example, the 'Only Fully Wheelchair Accessible' - that button should be highlighted in blue to provide an indication to the user of their selection.
5. When a user clicks the 'Only Fully Wheelchair Accessible' button, the app should respond by populating the screen with additional filters such as: 'Shopping', 'Food & Drink', etc..., for better ease of use and efficiency.

Testing schedule

Given the relatively small size of the system and number of test cases, our testing framework will not require scheduling. The idea is that the user can run these test cases at any time.

Test recording procedures

Under the direction of the Team Project Specifications document, our testing framework will produce and automatically open a professional-grade testing report detailing the test cases and the results of each test case execution. The Python unittest package produces a report by default after every run of the test cases (Figure 4)



```

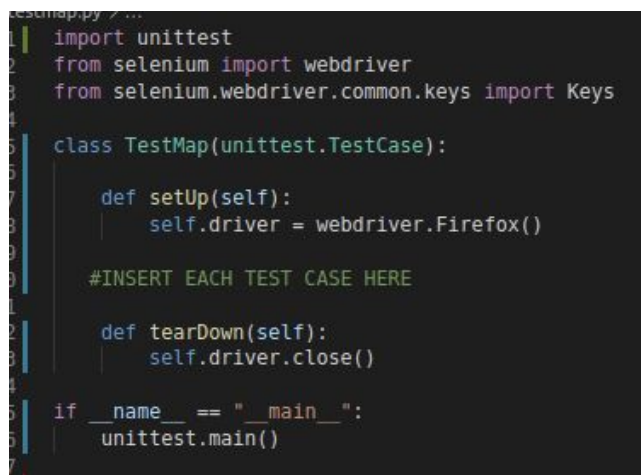
mattwalter@ubuntu:~/7-11$ /usr/bin/python3 /home/mattwalter/7-11/testmap.py
.
-----
Ran 1 test in 5.339s

OK

```

Figure 4: output from a successful test run

This contains the execution time of each test case and whether it passed or failed. We plan to use Python's built-in functionality for writing to and opening files in the implementation. We will include a line within each test case or within method in the driver class (Figure 5) that prints the necessary information to the report.



```

testmap.py > ...
1 | import unittest
2 | from selenium import webdriver
3 | from selenium.webdriver.common.keys import Keys
4 |
5 | class TestMap(unittest.TestCase):
6 |
7 |     def setUp(self):
8 |         self.driver = webdriver.Firefox()
9 |
10 |        #INSERT EACH TEST CASE HERE
11 |
12 |        def tearDown(self):
13 |            self.driver.close()
14 |
15 |        if __name__ == "__main__":
16 |            unittest.main()
17 |

```


Figure 5: the early-stage driver

Hardware and software requirements

The full procedure to build wheelmap-frontend locally and download the software requirements is outlined in the actual build sequence section of Chapter 1: Deliverable 1.

Short-list of software requirements to build the project

- Ubuntu Linux virtual machine
- A local clone of the wheelmap-frontend repository.
- Node.js 10.x or the latest version
- Node Package Manager (npm)
 - Default package manager for Node.js and the Javascript programming language
- Transifex
 - A web-based translation tool Wheelmap utilizes in order to seamlessly localize the user's experience (i.e. Transifex translates the Wheelmap app for Users based on location)

Procedure to set up software requirements for testing the project

1. Download python bindings for Selenium:
 - `sudo apt-get install python3-pip python-dev`
 - `pip3 install selenium`
2. Selenium requires a driver to interface with the chosen browser. Download geckodriver for your respective browser. Since we are using Linux systems and they come with Firefox, we defaulted to using that:
 - `wget`
<https://github.com/mozilla/geckodriver/releases/d>

```
ownload/v0.27.0/geckodriver-v0.27.0-linux64.tar.g  
z
```

- `tar -xvzf geckodriver*`
- `chmod +x geckodriver`
- `sudo mv geckodriver /usr/local/bin/`

Constraints

Wheelmap utilizes a paid third-party application (BrowserStack) in order to test its software. Since testing is done separately by the developers, we do not have access to the actual test cases they use.

As a result, our testing process is constrained to user acceptance testing. All test cases pivot around ensuring the graphical user interface (GUI) operates as expected when interacted with by the user.

System tests

1. On start up the program will ask for Cookie permissions
 - a. Oracle: popup appears with the text "Okay, let's go!"

```
testCase1.txt  
1  "id": 1,  
2  "requirement": elementRendered,  
3  "component": CookieButton,  
4  "input":  
5  elem1 = driver.find_element_by_class_name("button-continue-with-cookies")  
6  assert elem1.text == "Okay, let's go!"  
7  "output": PASS  
8  
9
```

2. On start up the program will ask for Cookie permissions
 - a. Oracle: popup appears with the text "Continue without Cookies"

```
testCase2.txt
1  "id": 2,
2  "requirement": "elementRendered",
3  "component": NoCookieButton,
4  "input":
5  elem2 = driver.find_element_by_class_name("button-continue-without-cookies")
6  assert str(elem2.text) not in driver.page_source
7  "output": PASS
8
9  |
```

3. Tapping the 'locate me' button when active, removes the blue circle from the screen
 - a. Oracle: the leaflet-interactive element is not displayed

```
testCase3.txt
1  "id": 3,
2  "requirement": "userLocationUpdate",
3  "component": leaflet-interactive,
4  "input":
5  elem3 = driver.find_element_by_class_name("leaflet-bar-part")
6  elemDot = driver.find_element_by_class_name("leaflet-interactive")
7  elem3.click()
8  assert str(elemDot) not in driver.page_source
9  "output": PASS
```

4. Tapping the 'locate me' button when inactive enables positioning and adds the user location dot.
 - a. Oracle: the leaflet-interactive element is displayed

```
testCase4.txt
1  "id": 4,
2  "requirement": "userLocationUpdate",
3  "component": leaflet-interactive,
4  "input":
5  elem4 = driver.find_element_by_class_name("leaflet-bar-part")
6  elemDot2 = driver.find_element_by_class_name("leaflet-interactive")
7  elem4.click()
8  assert str(elemDot2) in driver.page_source
9  "output": PASS
```

5. Entering “Halls Chophouse” in the search bar returns the address of the restaurant
 - a. Oracle: The address of Halls is displayed in the drop down menu as “King Street, 29424”

Chapter 3

Creating the Testing Framework

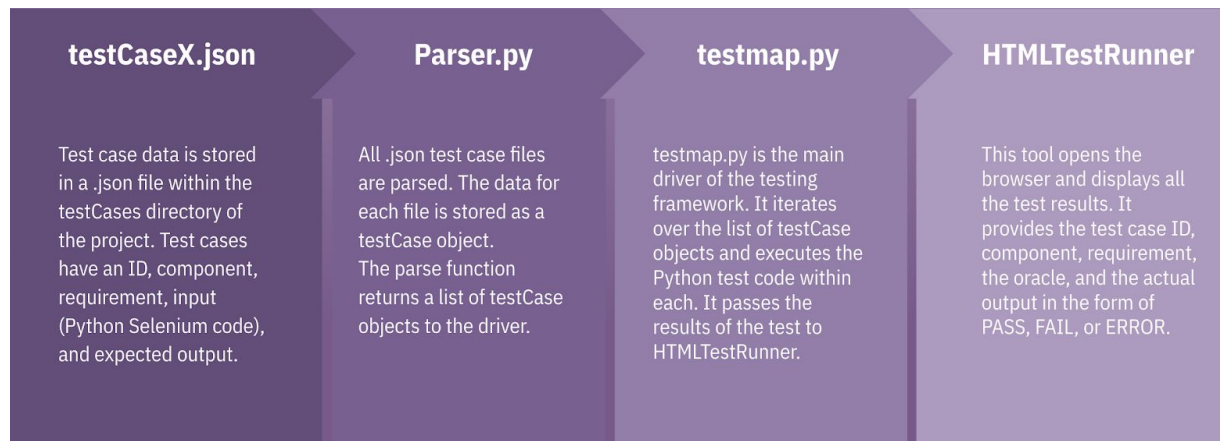
Deliverable task

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

Introduction

During this deliverable, we designed and built an automated testing framework to implement our test plan from the previous chapter. This is based on the Team Term Project Specifications document. To recap, we are testing the user interface of Wheelmap using Selenium and Python.

Framework overview



The Parser (Parser.py)

Key packages used:

- Glob

The parser is the first crucial piece of our framework. This program has a parse function which searches the testCases folder for all .json files. It then parses each file for the designated components of a test case (id, requirement, component, input, and oracle - in our case) and uses these attributes to create a testCase object.

The parse function ultimately appends each testCase object to a list which it returns.

```

6  def parse():
7      testList = []
8
9      for path in glob('./TestAutomation/testCases/*.json'): # loop over .json files in the cwd
10         with open(path) as f:
11             data = json.load(f) # open the json file
12             test = testCase(data['id'], data['requirement'], data['component'], data['input'], data['output'])
13             testList.append(test)
14         return testList

16 class testCase:
17
18     def __init__(self, id, req, component, input, oracle):
19         self.id = id
20         self.req = req
21         self.component = component
22         self.input = input
23         self.oracle = oracle

```

Figure 6: parse function and testCase class

The driver (testmap.py)

Key packages used:

- Selenium
- Unittest - a unit testing framework that is part of the standard Python library
- HTMLTestRunner

Our driver program first calls the parser to get the returned list of test case objects. It contains a class “TestMap” which then utilizes the Selenium and Unittest packages to perform the series of actions within the user interface given by the input of the test case. These are performed on the Firefox browser running Wheelmap locally.

```

9 class TestMap(unittest.TestCase):
10
11     def setUp(self):
12         self.driver = webdriver.Firefox()
13         self.driver.get("http://localhost:3000")
14
15     def test_function(input):
16         def test(self):
17             exec(input) in globals(), locals()
18             return test
19
20     def tearDown(self):
21         self.driver.close()
22
23
24 if __name__ == "__main__":
25     testmap = Parser.parse()
26
27     for test in testmap:
28         test_func = TestMap.test_function(test.input)
29         setattr(TestMap, 'test_{0}'.format(test.id), test_func)
30
31     # generate the HTML report
32     unittest.main(testRunner=HtmlTestRunner.HTMLTestRunner(output='../reports', report_name='testReport', open_in_browser=True))
33
34
35 1 #!/bin/sh
36 2
37 3 cd ..
38 4 rm -rf temp
39 5 mkdir temp
40 6 rm -rf reports
41 7 cd scripts
42 8
43 9 python3 testmap.py -v

```

Figure 7: testmap.py and runAllTests.py

The report

To create the report, we used a package called HTMLTestRunner. This package fits well with our framework as it integrates with Unittest, which we utilized as a framework for our test cases. It serves as a Unittest runner that saves test results in a user-friendly format within .html files.

Unittest Results

Start Time: 2020-11-05 09:43:35

Duration: 61.68 s

Summary: Total: 6, Pass: 6

__main__.TestMap		Status
test_1		Pass
test_2		Pass
test_3		Pass
test_4		Pass
test_5		Pass
test_function	test_function	Pass

Total: 6, Pass: 6 -- Duration: 61.68 s

Figure 8: report output example

Test cases

Test case #1: Finds the “accept cookies” button

```

1  {
2      "id": "1",
3      "requirement": "elementRendered",
4      "component": "CookieButton",
5      "input": "elem = self.driver.find_element_by_class_name
              (\"button-continue-with-cookies\")\nassert elem.text == \"Okay, let's go!\"",
6      "output": "PASS"
7  }

```

Test case #2: Finds the “continue without cookies” button

```

1  {
2      "id": "2",
3      "requirement": "elementRendered",
4      "component": "NoCookieButton",
5      "input": "elem = self.driver.find_element_by_class_name
              (\"button-continue-without-cookies\")\nassert elem.text == \"Continue
              without cookies\"",
6      "output": "PASS"
7  }

```

Test case #3: Turns the location button off

```

1  {
2      "id": "3",
3      "requirement": "userLocationButtonRendered",
4      "component": "leaflet-interactive",
5      "input": "elem1 = self.driver.find_element_by_class_name
              (\"button-continue-with-cookies\")\nelem1.click()\nelem = self.driver.
              find_element_by_class_name(\"leaflet-bar-part\")",
6      "output": "PASS"
7  }

```

Test case #4: Turns the location button on

```

1  {
2      "id": "4",
3      "requirement": "userLocationUpdate",
4      "component": "leaflet-interactive",
5      "input": "elem1 = self.driver.find_element_by_class_name
               (\"button-continue-with-cookies\")\nelem1.click()\nelem = self.driver.
               find_element_by_class_name(\"leaflet-control-zoom-in\")",
6      "output": "PASS"
7  }

```

Test case #5: Tests the search function

```

1  {
2      "id": "5",
3      "requirement": "searchBar",
4      "component": "search-input",
5      "input": "elem1 = self.driver.find_element_by_class_name
               (\"button-continue-with-cookies\")\nelem1.click()\nelem5 = self.driver.
               find_element_by_class_name(\"search-input\")\nelem5.send_keys(\"Halls
               Chophouse\")\nelem5.send_keys(Keys.RETURN)",
6      "output": "PASS"
7  }

```

How-to guide

We have added some dependencies with this deliverable. First, please install the following:

- `pip3 install html-testRunner`
- `pip3 install glob`
- `pip3 install selenium`

- `wget`
`https://github.com/mozilla/geckodriver/releases/download/v0.27.0/geckodriver-v0.27.0-linux64.tar.gz`
- `tar -xvzf geckodriver*`
- `chmod +x geckodriver`
- `sudo mv geckodriver /usr/local/bin/`

Next, get Wheelmap up and running by executing the following commands within the project folder:

- `cp .env.example .env`
- `npm install`
- `npm run dev`

Lastly, simply navigate to the top-level directory, TestAutomation, and execute “./scripts/runAllTests.py”. This will ultimately open the .html report of the test case results.

Chapter 4

Completing the Test Case Suite

Deliverable task

*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.*

Introduction

In chapter 4, we developed all 25 test cases for the user interface for Wheelmap. We also made improvements based on the feedback we received on deliverable 3, particularly about making our output report more verbose.

Updates to report

During deliverable 3, our test case output looked like Figure 9.

Unittest Results

Start Time: 2020-11-05 09:43:35

Duration: 61.68 s

Summary: Total: 6, Pass: 6

__main__.TestMap	Status
test_1	Pass
test_2	Pass
test_3	Pass
test_4	Pass
test_5	Pass
test_function	Pass

Total: 6, Pass: 6 -- Duration: 61.68 s

Figure 9: test case output from deliverable 3

Dr. Bowring gave us the feedback to make this report more verbose, meaning to add all parts of the test case (component, requirement, output) to the output. We are using a package called HTMLTestRunner to configure the output to our unittest test cases; initially we were just using the default template. To update the columns of the output table, we had to create our own template to pipe to HTMLTestRunner. It required a new template engine none of us had ever worked in called Jinja2.

```

{%- for test_case in tests_results %}
{%- if not test_case.subtests is defined %}
<tr class='{{ status_tags[test_case.outcome] }}'>
  {%- if not test_case.test_id.split(".")[1] == "test_function" %}
  <td class="col-xs-5">test_{{ testCase_list[loop.index0].id }}</td>
  <td> {{ testCase_list[loop.index0].component }}</td>
  <td> {{ testCase_list[loop.index0].req }}</td>
  <td> {{ testCase_list[loop.index0].output }}</td>
  {%- endif -%}

```

Figure 10: template.html

In short, we started with the template provided by the HTMLTestRunner Github. Then, we added new columns and as we looped through creating each row, used the loopindex to grab the correct requirement, component, or output to add.

to the column-row combination.

Start Time: 2020-11-17 11:49:09

Duration: 253.16 s

Summary: Total: 26, Pass: 26

TestMap for WheelMap	Component	Requirement	Output	Status
test_11	CONTACT-nav-link	Link redirect	PASS	Pass
test_7	Claim	text rendered	PASS	Pass
test_15	ADDPLACE-nav-link	Link redirect	PASS	Pass
test_16	ImproveThisMap-nav-link	Link redirect	PASS	Pass
test_21	OpenStreetMap	SubComponent click	PASS	Pass
test_20	MapBox	SubComponent click	PASS	Pass
test_17	sozialhelden-logo	Link redirect	PASS	Pass
test_9	NEWS-nav-link	Link redirect	PASS	Pass
test_22	User-ZoomOut	ZoomOut	PASS	Pass
test_23	ac-marker-yes	elementRendered	PASS	Pass
test_4	leaflet-interactive	userLocationUpdate	PASS	Pass
test_5	search-input	searchBar	PASS	Pass
test_25	ac-marker-no	elementRendered	PASS	Pass
test_24	ac-marker-limited	elementRendered	PASS	Pass
test_10	PRESS-nav-link	Link redirect	PASS	Pass
test_12	IMPRINT-nav-link	Link redirect	PASS	Pass
test_1	CookieButton	elementRendered	PASS	Pass
test_6	logo	elementRendered		Pass
test_14	EVENTS-nav-link	Link redirect	PASS	Pass
test_2	NoCookieButton	elementRendered	PASS	Pass
test_13	FAQ-nav-link	Link redirect	PASS	Pass
test_18	User-ZoomIn	ZoomIn	PASS	Pass
test_3	leaflet-interactive	userLocationButtonRendered	PASS	Pass
test_8	GetInvolved-nav-link	Link redirect	PASS	Pass
test_19	MapBox-wordmark	SubComponent click	PASS	Pass
Pass				

Total: 26, Pass: 26 -- Duration: 253.16 s

Figure 11: test case output from deliverable 4

TestMap Component Test

Start Time: 2020-11-17 12:01:19

Duration: 21.89 s

Summary: Total: 3, Pass: 1, Fail: 1, Error: 1

TestMap for WheelMap	Component	Requirement	Output	Status	
test_1	CookieButton	elementRendered	PASS	Fail	Hide
<p>AssertionError:</p> <p>Traceback (most recent call last): File "./scripts/testmap.py", line 17, in test exec(input) in globals(), locals() File "", line 2, in AssertionError</p>					
test_2	NoCookieButton	elementRendered	PASS	Error	Hide
<p>SyntaxError: invalid syntax (, line 2)</p> <p>Traceback (most recent call last): File "./scripts/testmap.py", line 17, in test exec(input) in globals(), locals() File "", line 2 assertt elem.text == "Continue without cookies" ^ SyntaxError: invalid syntax</p>					
<div>Pass</div>					

Total: 3, Pass: 1, Fail: 1, Error: 1 -- Duration: 21.89 s

Figure 12: test case output Failure and Error modes

```

24 if __name__ == "__main__":
25
26     testsmap = Parser.parse()
27
28     for test in testsmap:
29         test_func = TestMap.test_function(test.input)
30         setattr(TestMap, 'test_{0}'.format(test.id), test_func)
31
32     template_args = {
33         "testCase_list": testsmap
34     }
35
36     unittest.main(testRunner=HtmlTestRunner.HTMLTestRunner(template='./scripts/template.html',
37         template_args=template_args, output='../reports', report_name='testReport',
38         open_in_browser=True, report_title='TestMap component test'))

```

*Figure 13: testmap.py (driver) updated passing in testsmap list into
HtmlTestRunner*

Reflection

This deliverable, we got together as a team a few times to focus on the task at hand. Learning about Jinja2 to reformat the report was an unexpected but valuable experience of working with a new template engine. We also had an interesting new software engineering experience. After spending a few hours working on our modified Jinja2 template, we finally got the report outputting the way we wanted. When we met the next day to put the finishing touches on this deliverable, we realized that the template had not saved! Fortunately, we were able to put the template back together in a fraction of the time it first took us.

Chapter 5

Injecting Faults into the Code

Deliverable 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.

Introduction

In chapter 5, we injected five faults into the Wheelmap-frontend code (not the test cases themselves), causing five of our test cases to fail. The objective of this exercise was to make at least five, but not all, test cases fail. This was a unique challenge for our team because we are testing a graphical user interface. Many of the components that we wanted to adjust to fail our test cases would break the whole application. Below are the changes we made, including the path of the changed file, the line number altered, the update to that line, and the test case impacted:

Changes and associated test cases

	Path to File	Line #	Line Changes	Test Case Impacted
1	/src/components/Onboarding/Onboarding.js	44	const startButtonCaption = t`Okay, let's gooo!`;	TestCase1
2	/src/components/Onboarding/Onboarding.js	46	const skipAnalyticsButtonCaption = t`Continue without cookiesss`;	TestCase2
3	/src/components/Map/addLocateControlToMap.js	45	title: t`Show me where I ammm`,	TestCase3
4	/src/components/Map/addLocateControlToMap.js	1103		TestCase16
5	/src/components/Map/addLocateControlToMap.js	1086		TestCase17

Figure 14: Fault injection table

To inject faults into the code, simply locate the file in the "Path to File" column, locate the line number, and replace the line with the text given in the corresponding "Line Changes" entry. Then, recompile the project to see the faults in action.

Explanation of Changes

- **Test Case 1:** We altered the text on the Continue With Cookies button as it was defined within the Oboarding.js file. The difference between this text and the original text in the test case resulted in an assertion error, failing the test.
- **Test Case 2:** This change was very similar to the one in test case 1. We altered the text for the Skip Analytics button within the code while leaving it with the original text in the text case. This again caused an assertion error and a failed test.
- **Test Case 3:** Here we changed the text associated with the location control button that recenters the user's map to his or her location. Since the test case checks for the correct text, this causes it to fail.
- **Test Case 16:** This test case checks that the correct link opens when the user clicks Improve This Map. We altered the associated link within the addLocateControlToMap.js file so that the assertion statement checking the link would fail.
- **Test Case 17:** This test case change used the same logic as that in test case 16, but altered a different link.

TestMap Component Test

Start Time: 2020-11-24 11:20:28

Duration: 49.42 s

Summary: Total: 6, Pass: 1, Fail: 5

TestMap for WheelMap	Component	Requirement	Oracle	Status
test_001	ImproveThisMap-nav-link	Link redirect	PASS	Fail View
test_002	sozialhelden-logo	Link redirect	PASS	Fail View
test_003	CookieButton	elementTextRendered	PASS	Fail View
test_016	NoCookieButton	elementTextRendered	PASS	Fail View
test_017	leaflet-interactive	userLocationButtonRendered	PASS	Fail View
<div>Pass</div>				
Total: 6, Pass: 1, Fail: 5 -- Duration: 49.42 s				

Figure 15: The failed tests displaying in the test case report

Experience Report

The nature of our project made this an interesting experience for our team. When we were first working together to brainstorm how we could make our existing test cases fail, we realized that with certain changes, the program would present an error in the user interface that our test case would either circumvent (and still pass) or halt on (and present an error rather than a fail). We learned that failure of the assert statement within the test case is truly what causes the test to fail. Our objective then became finding specific enough code items we could change so that a test case assert statement would fail, but the whole application would stay in-tact.

Chapter 6

Conclusions

Team evaluation

All in all, our experience working on a team based software project was extremely positive. We were fortunate enough to have a game plan from the beginning. We stuck with it, making it work along the way. As a team we worked well together. We divided the work hours very evenly and made everyone feel intellectually challenged while producing a project we are proud of. If we had to start the project over again, we would undoubtedly change things to better suit our code base's needs and potentially test different aspects of the web server.

Lessons learned

This project increased our knowledge in several areas practical to software engineering.

First, we were exposed to the Git workflow throughout the course of the project. We had a practical experience (explained in Chapter 4) in which we lost some of our progress due to a missing step in the Git workflow. This was a part of the project we had spent several hours working on the day prior, so losing that file was a hard but valuable software engineering lesson. Of course, Git is a tool that we will all continue to use throughout our careers. Learning about it, even through trial and error, was extremely valuable.

Building one key component of our framework proved an excellent learning experience. We built a parser that fetches files and parses their contents into Python test case objects. For our instance, this was the best way to conform to the assignment requirement of not “pre-loading” test cases.

Our project was also unique in that it allowed us to work with several powerful packages. The most important to our project was Selenium, an automation library for web browser activity through which we automated the “input” actions of our test cases. When it came to printing our output to the browser, we were able to utilize a tool called HTMLTestRunner, a test runner which integrates well with the test case package we used (unittest) to generate professional grade reports. To fit the generated reports to the standard of the specifications, we had to jump into editing the standard Jinja-2 web template for HTMLTestRunner. None of us had ever worked with this engine, so this proved another challenging but valuable experience. Additional tools we gained experience with throughout the project include python package manager (pip) and node package manager (npm).

These lessons will be invaluable to our careers in the software engineering field.

Project weaknesses

The biggest weakness of our project is that its structure does not lend to scalability. Prior to execution, our framework loads all of our test cases into memory. Our framework uses one web server and for each new test it runs, it opens up a new browser window. This is a time intensive process. At the current size of our test case set (25 cases), our script takes approximately four minutes to run and open up the .html report in the browser.

If exponentially more test cases were added, our framework may not hold up to the memory requirements. The time requirements alone would make it unrealistic to run this kind of testing framework on such a large test case set. We would have to make changes to the way that our server loads and runs the test cases to increase efficiency. However, for a relatively small set of test cases that test the basic functionality of a graphical user interface, this framework does the trick.

Project evaluation

In general, we thought that the expectations for assignments were very clearly laid out on the class website. We were never left with many questions about how to complete the assignments. Our only critique of the project as a whole was that we wish the list of open-source projects was better vetted. We came across a number of projects that were no longer maintained or did not fit the specifications of the team term project well. It was a good experience to build some such projects other than our top choice. However, we were afraid that we could get far along with any project, then run into a critical issue. In future sections, it could be helpful to provide a list of projects students have had success with in the past.