College of Charleston

Team Term Project Final Report

Team3

Matthew Anuszkiewicz, Benjamin Duke, Tino Pimentel

Software Engineering CSCI 362

Jim Bowring

November 21st, 2019

Table of Contents

Introduction

Here at the College of Charleston, many of us have finished working on a project for our CSCI 362 Software Engineering course. All teams involved in the course had to select an open source H/FOSS project to proceed with practicing what we have learned throughout the course on. OpenAQ (Open Air Quality) is the H/FOSS project our group (Team3) decided to work on. OpenAQ is a software which gathers data from thousands of real-life sensors which sample the air quality in many countries throughout the world. These sensors collect data using many different units of measurement. We have been using the development technique known as Test-Driven Development for the strategy in which we develop tests. We design our set of tests for the project based on a set of units. The unit we are testing in particular for our project is a part of the OpenAQ library known as OpenAQ-Fetch, a tool used to collect data for the OpenAQ platform. The main goal of this project is to learn how to design a testing framework, including an architectural description, for complex software projects. Through implementing our own designed automated testing framework, we are to learn many techniques for developing a testing framework. The programming languages we are using, in particular for OpenAQ, include BASH, HTML and JSON. We utilized these languages to execute tests within OpenAQ-Fetch's server which is built using Node.js. Node.js is a JavaScript run-time environment that comes with a HTTP module that provides a set of functions and classes for building a HTTP server. OpenAQ-Fetch has a function in particular that we are testing called "MakeSourceFile( )" which reads in input from the sensors as a JSON file that OpenAQ collected. OpenAQ-Fetch then analyzes its results from within the HTTP server for a valid reading of the sensor data. Our group is testing over twenty-five test cases in which we send JSON files of different parameters to the HTTP server through Node.js and then comparing to our own oracle (Test Driven Development artifact) to see if the "MakeSourceFile( )" method is executing without any errors or exceptions.

Chapter 1

Introduction

For our first deliverable, our group needed to evaluate several H/FOSS projects to work on designing an "Automated Testing Framework" for. The H/FOSS project we choose will determine how we design our testing framework. The criteria involved for deciding on which project to move forward with include:

- Code base is designed to compile and run on a Linux machine.
- The code base is available within GitHub.
- The code base can compile and run within a Linux virtual machine.

Once we were able to decide on our H/FOSS project, and compile the project. We then needed to run what-ever existing tests the software had and collect the results. In addition to the previous tasks, we were instructed to complete a team exercise which involved writing the contents of any folder a file called "myList.sh" was executed into an HTML file.

Accomplishments

During project development for Deliverable 1, our group has decided which H/FOSS project as a candidate we would proceed with. Our group has already attempted to compile the code base to our best-known candidate, Epidemiological Modeler (STEM). We also successfully completed the team exercise and are able to execute "myList.sh" within a terminal using the BASH programming language command, "./myList.sh".

Learning Outcomes

Our team has learned a basic understanding of how to write code in BASH, and HTML. How to cooperate and communicate as a team to accomplish simple and semi-complex goals. We also learned how to operate within a Linux environment when dealing with a complex software. Lastly, we learned how to formally build a code base from a given repository.

Problems Experienced

During the writing of our wiki for our project's deliverable 1, we received feedback from our instructor to host the wiki on GitHub instead of Fandom. During the compiling of our best candidate's code base, the system would continuously report errors to where we could not execute any build-in tests available with the software. Some of the plugins in the tutorial for compiling the project are outdated, and presumably covered by newer plugins. However, upon attempting to launch STEM, a "Framework Error" stopped the launch. The developer build is still a work in progress. The standalone STEM release should work for a number of pre-programmed scenarios, however the program hangs up when starting a scenario. A screenshot of one of the errors is provided below.

Going forward

Ultimately, once one machine can properly install the packages and sort through the dependencies of STEM that contain bugs. Then we can begin developing scripts to start testing our project. If we cannot get the software running properly, then we will need to proceed to compile our second candidate, OpenAQ. In preparation for our next deliverable we need to write a test plan for our automated testing framework.

Team exercise source code is linked below:

Team Exercise: https://github.com/csci-362-01-2019/Team3/blob/master/myList.sh

Candidate breakdowns provided below:

Candidate 1: Epidemiological Modeler (STEM) - Language used: Java

- Simulates the spread of various infectious diseases throughout different populations.
- Huge library, will take some time to learn.
- Well documented with tutorials on how to get started.

Candidate 2: Open Air Quality (OpenAQ) - Languages used: JavaScript

- Reports how fresh the air quality is in a given zip code using many measurements.
- The measurements are collected from many sensors across the world.
- The software is decently documented and lengthy.
- Language will take some time to understand in order to work with project.

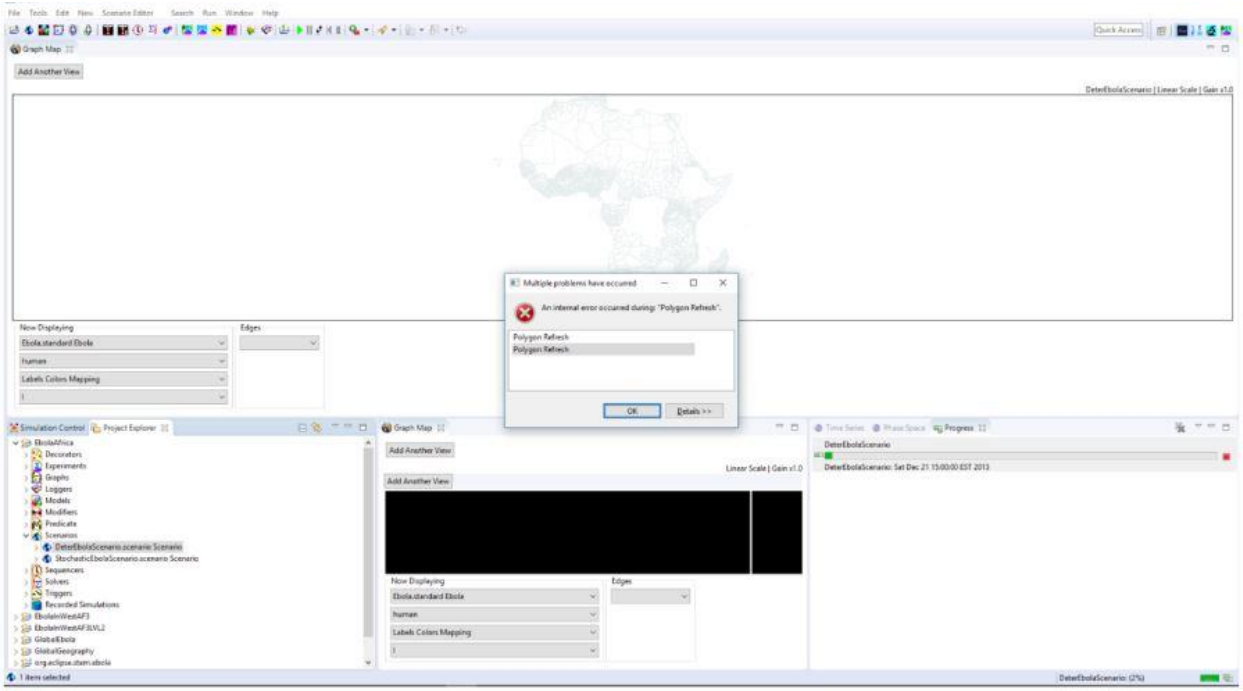
Candidate 3: Nightscout - Languages used: C, Java, HTML

- CGM (Continuous Glucose Monitoring) software.
- Compatible on many Operating Systems such as Windows, IOS, Android, Linux.
- Setup guarantees errors with too many dependencies to properly demonstrate from a fresh setup.
- Formal setup guides, however error bound.
- Many applications developed for the Web Browser, Android, IOS, and even several smartwatches.

Candidate 4: Drone 4 Dengue - Languages used: JavaScript, HTML

- Image processing software that detects mosquito breeding sites.
- The images are grabbed by a drone in the area.
- Not very well documented.
- Project appears to be incomplete.

A screenshot of an error with STEM can be seen below:



Chapter 2

Introduction

Since our previous Deliverable, our group decided to switch the HFOSS project we focused on. We decided to switch to OpenAQ because STEM included many problems for our group including:

- Issues finding all packages of the source code.
- Building what little of the source code we actually could obtain.
- Not being able to successfully build the project on multiple machines running Ubuntu Linux.

With a lack of running software to develop or run tests on, our group could not continue making progress within the overall scheme of the project. Therefore, our group has decided to work with OpenAQ as our new #1 candidate HFOSS project. OpenAQ includes:

- All source code needed to properly build from scratch.
- Guides to build the project from the source code.
- Project being written in JavaScript.
- Successfully built and inspected on a few machines already (more on this later).

Accomplishments

During project development for Deliverable 2, our group has accomplished writing a test plan, which includes the hierarchy of how we are to proceed with testing different aspects of OpenAQ throughout the rest of the semester. Our team has also written how we would test certain operations within OpenAQ, 5 out of 25 is what we wrote so far.

Learning Outcomes

Our team has learned how to develop a test plan for a detailed method of testing different parts of the software from bottom to top. The work done helped develop a further understanding of how to write test cases for a complex software which includes multiple components and interfaces.

Problems Experienced

Even though the guides for building OpenAQ appear to be well written, the steps to successfully building OpenAQ sometimes appear to be out of order. At particular steps, a software package would not install because prerequisite software was instructed to be installed afterwards. Installing all software components when needed resolved many of these issues.

Going forward

The next goal is to keep uncovering more information of how the software functions in order to develop further relevant test cases. As our team learns more software engineering principles to

apply, we can further strategize our resources efficiently towards completing future Deliverables for OpenAQ.

Below is the Test Plan for the OpenAQ project.

The Openaq project is an open-source project that gets air quality data from different cities and countries and makes them available to the public. The pollutants Openaq tracks are PM10, PM2.5, sulfur dioxide (SO2), carbon monoxide (CO), nitrogen dioxide (NO2), ozone (O3), and black carbon (BC). It gathers this data from official-level outdoor air quality source. The readings must meet the criteria of Openaq-fetch in order to be added to the database.

This is the test plan for openaq-fetch, the main data ingest pipeline for the OpenAQ project.

The Testing Process

During the Testing of OpenAQ-fetch, an order of phases will occur. The first will be Unit-Testing where individual operations are tested with a set of test cases to guarantee the operations meet required specifications. The second phase will be Component-Testing where multiple operations are tested together using an interface (forming a component) to guarantee that a particular set of criteria is met. The last phase will be the overall System-Testing in which a select set of components are integrated together and then tested for expected conditions to be met from the software.

Requirements Traceability

The software is required to function as expected which entails:

- The software performs under expected conditions
- The software receives all necessary parameters within every operation during any set of tests
- The interactions between the components during system testing must match with what the oracle predicts.
- A report is made after a set of automated tests have completed.

Tested Items

An operation within OpenAQ known as, MakeSourcefromData(), which adds air quality data to the database. The operation uses parameter unit values such as coordinates, dates, value of ppv (particles per volume), and units of measurement.

Testing Schedule

The schedule will be as follows:

- The first set of Unit-Tests will occur during the development/understanding of code for a couple of weeks from October 1st to October 29th.

- The next set of testing will include the development and execution of an automated testing framework for 25 test cases from October 29th to November 12th.
- The final set of testing involves fault testing where components within the framework will be tested with malicious input for expected results from November 12th to November 19th.

Test Recording Procedures

The software will be running tests from a script file that records the results of the tests into text file with a particular format depending on what part of the system is being tested. The results are then compared to the oracle's text file for correctness & expected output. Lastly, a report is made onto an HTML file of the results from testing our project.

Hardware and Software Requirements

Software & Hardware Requirements include:

* Windows, Mac, Linux system

* Approx 200MB of free disk space

* At least 1.8GB free memory

* 2 core CPU is recommended

* node.js >= 8.6

* Either of:

  * docker or

  * postgresql >= 10 with postgis >= 2 extensions.

Constraints

Constraints include:

- **Lack of Time** due to the testing portion of the project being limited to 2 months.
- **Lack of Staff** due to amount of students working on the project being limited to 3 members with diverse tasks.
- **Lack of Knowledge** due to every student working on the project still learning methods for testing.

System Tests

Test cases will include tests ran through scripts upon the operations of OpenAQ-fetch for pass/fail conditions. Tests will be ran on the inputs for several particular operations for stability, and expected output. Tests will be ran on the outputs of several operations for correctness. Tests

will be ran on order of execution for the system components as well as the overall system for expected goal completion.

Each test case is created in javascript and run from the command line using: node index.js --dryrun 'test'

--dryrun, -d Run the fetch process but do not attempt to save to the database and instead print to console, useful for testing.

Test case 1: Input: parameter: 'pm25', unit: 'ppq', //Error: Should be ppm value: 10, location: 'test1', coordinates: { latitude: -20, longitude: 34 }, country: 'US', city: 'Test', sourceName: 'Test', mobile: false, sourceType: 'government', attribution: [{ name: 'test', url: 'http://test.case' }], averagingPeriod: { value: 1, unit: 'hours' }

Expected Output: 'instance.unit is not one of enum values: µg/m³,ppm'

Test case 2: Input: parameter: '01', //Error: Not a valid parameter unit: 'ppm', value: 10, location: 'test2', coordinates: { latitude: -20, longitude: 34 }, country: 'US', city: 'Test', sourceName: 'Test', mobile: false, sourceType: 'government', attribution: [{ name: 'test', url: 'http://test.case' }], averagingPeriod: { value: 1, unit: 'hours' }

Expected Output: 'instance parameter is not one of enum values: pm25,pm10,no2,so2,o3,co,bc'

Test case 3: Input: parameter: 'no2', unit: 'ppm', value: Q, //Error: Value must be a number location: 'test3', coordinates: { latitude: -20, longitude: 34 }, country: 'US', city: 'Test', sourceName: 'Test', mobile: false, sourceType: 'government', attribution: [{ name: 'test', url: 'http://test.case' }], averagingPeriod: { value: 1, unit: 'hours' }

Expected Output: 'instance value is not of a type(s) number'

Test case 4: Input: parameter: 'pm25', //Ideal case unit: 'ppm', value: 10, location: 'test4', coordinates: { latitude: -20, longitude: 34 }, country: 'US', city: 'Test', sourceName: 'Test', mobile: false, sourceType: 'government', attribution: [{ name: 'test', url: 'http://test.case' }], averagingPeriod: { value: 1, unit: 'hours' }

Expected Output: 'Data source accepted.'

Test case 5: Input: parameter: 'pm25', unit: 'ppm', value: 10, location: 'test5', coordinates: { latitude: -20, longitude: 34 }, country: , //Error: Country field required city: 'Test', sourceName: 'Test', mobile: false, sourceType: 'government', attribution: [{ name: 'test', url: 'http://test.case' }], averagingPeriod: { value: 1, unit: 'hours' }

Expected Output: 'instance requires property "country"'

Chapter 3

Introduction

Since our previous deliverable, our group has worked on implementing our automated testing framework, while developing 15 of 25 test cases.

The goals for this deliverable were to:

- Rework the test plan written during Deliverable 2 (Chapter 2).
- Develop an architectural diagram or description of our testing framework.
- Document a full how-to on deploying our test environment and project.
- Developing at least 5 of 25 eventual test cases within OpenAQ-Fetch.

Plenty of time has been spent writing code to fulfill the core requirements of our automated testing framework.

Accomplishments

During project development for Deliverable 3, our group has accomplished rewriting the test plan previously mentioned, which includes the hierarchy of how we are to proceed with testing different aspects of OpenAQ throughout the rest of the semester. 15 test cases were written for OpenAQ-Fetch to execute. The automated testing framework was written in the BASH programming language. Documentation was provided for deploying and executing a full clone of the project, as well as our automated testing code. Lastly, an HTML file was written to report the outcomes of the testing framework based on the test cases used within OpenAQ-Fetch.

Learning Outcomes

Our team has learned how to develop a proper test. Our team has learned how to write code in BASH, as well as HTML. The work done helped develop a further understanding of how to write proper test cases for a complex software which includes multiple components and interfaces. Knowledge of a “Layered Architecture” was discovered for designing the description of our automated testing framework.

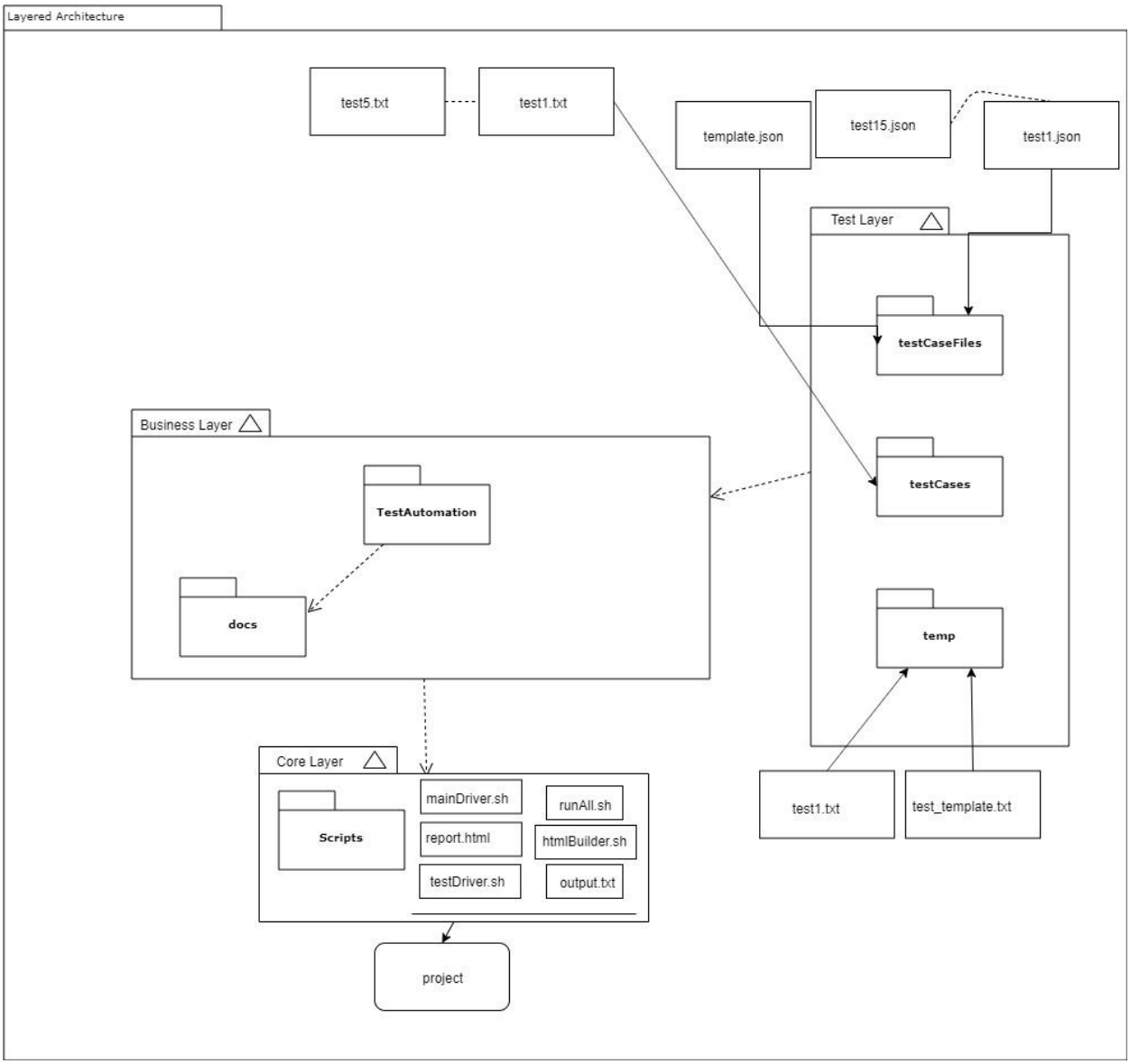
Problems Experienced

During development of the BASH code for OpenAQ-Fetch, we were having trouble discovering the proper syntax and method to parse and execute a string of arguments from a separate text file which contained our test cases.

Going forward

The next goal, per specification of our test plan, is to one hundred percent finish the design and implementation of our testing framework. As well as finish developing a full set of 25 test cases for OpenAQ-Fetch.

First version of our Layered Architecture can be found below


Layered Architecture
test5.txt
test1.txt
template.json
test15.json
test1.json
Test Layer
testCaseFiles
Business Layer
TestAutomation
testCases
docs
temp
Core Layer
mainDriver.sh
runAll.sh
Scripts
report.html
htmlBuilder.sh
testDriver.sh
output.txt
test1.txt
test_template.txt
project

Chapter 4

Introduction

Since our previous deliverable, our group has continued working on implementing our automated testing framework while developing all of our 25 test cases. As we ramp up towards the class' "Final Presentation Day" we are making efforts towards finalizing our documentation, and presentation.

The goals for this deliverable were to:

- Complete the design and implementation of our testing framework (Chapter 2).
- Start working on the poster for our project.
- Write a report using HTML.
- Developing all 25 test cases within OpenAQ-Fetch.

Plenty of time has been spent writing code to fulfill the core requirements of our automated testing framework. Additional code was written in HTML to finalize the results of the automated testing framework into a neat report.

Accomplishments

During project development for Deliverable 4, our group has accomplished writing the rest of our test cases for OpenAQ-Fetch to execute within the automated testing framework we implemented last deliverable. The final poster has been started, including basic information about what OpenAQ is, as well as what we are doing with it. The architectural diagram for describing the automated testing framework has been updated within draw.io. Lastly, our previously written HTML report was updated to make the test case numbers, and labeling more comprehensive to everyone. Added functionality was included in the HTML report by making the tables presented able to sort themselves, adding further comprehensiveness to the person viewing the report.

Learning Outcomes

Our team has gained much patience for testing and debugging code that is producing errors upon minor changes. Our team has improved on writing code in HTML, and BASH. The work done helped develop a further understanding of how to design and implement a comprehensive report without breaking the project. Proper usage of draw.io (online) was made familiar through implementing the Layered Architecture described last chapter.

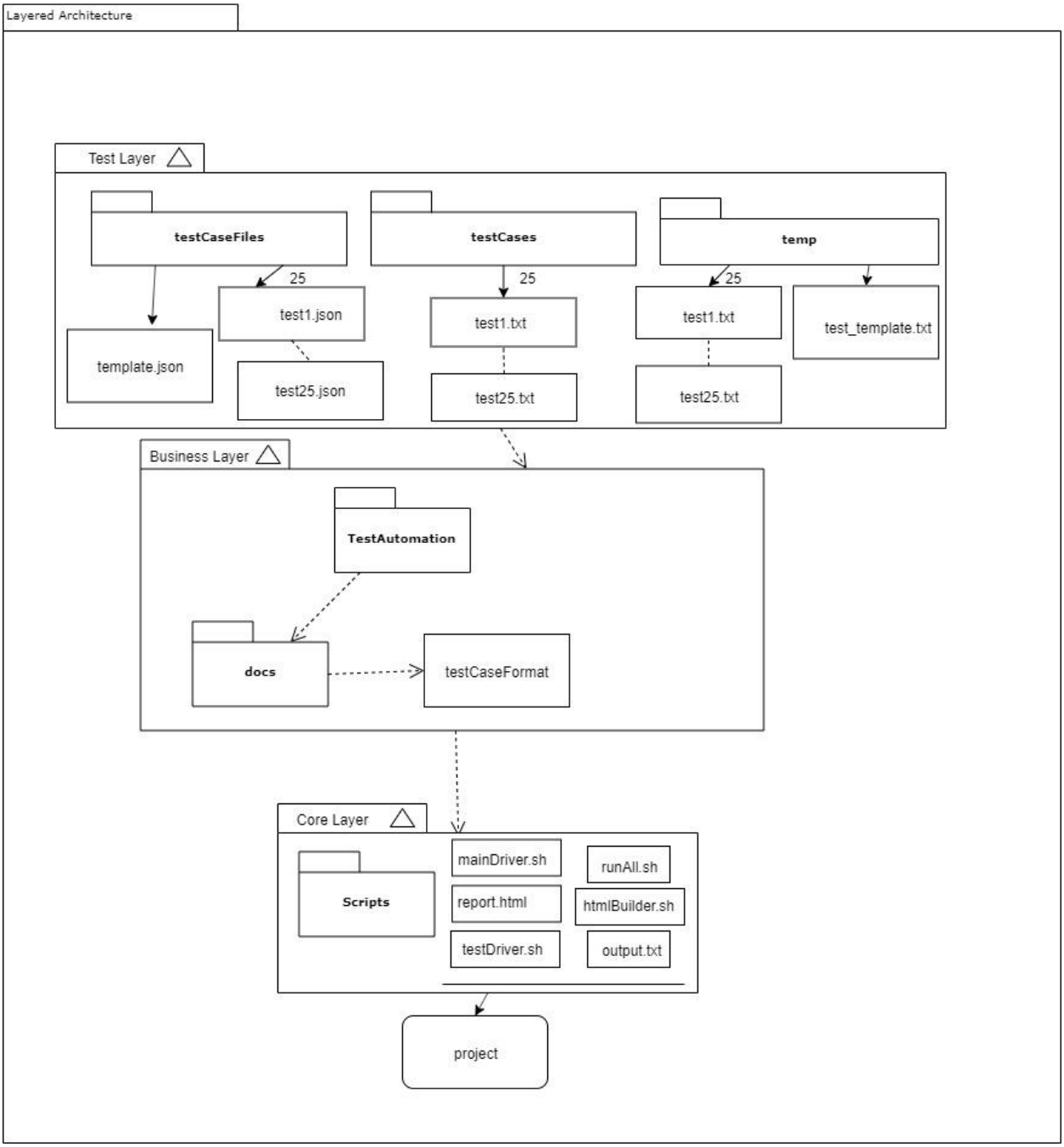
Problems Experienced

During development of the HTML code for our report of the testing output, we were having trouble writing code that could produce a decent aesthetic for labels, and tables. During our initial solution for making the report comprehensive to view, some of the tests failed when previously passing within the testing framework.

Going forward

The next goal, per specification of our test plan, is to implement fault testing where components within the framework will be tested with malicious input for expected results. As well as finish the project poster and presentation for our “Final Presentation Day”.

Second version of our Layered Architecture can be found below



Chapter 5

Introduction

Since our previous deliverable, our group has been working on clarifying our automated testing framework based on previous feedback. As we ramp up towards the class' "Final Presentation Day" we are making efforts towards finalizing our documentation, and presentation.

The goals for Deliverable 5 were to:

- Incorporate 5 injection faults into the source code of our /HFOSS project.
- Continue working on the poster for our project.
- Include more clarification in our HTML script.
- Make the testing framework documentation more comprehensive.
- Rewrite Chapter 1 of the final report.

Plenty of time has been spent writing code to fulfill the core requirements of our automated testing framework. Additional code was written in HTML to finalize the results of the automated testing framework into a neat report.

Accomplishments

During project development for Deliverable 5, our group has accomplished creating an Activity Diagram for our project using draw.io (online). The activity diagram has been added to the final poster. The architectural diagram for describing the automated testing framework has been updated within draw.io and added to the poster as well. Chapter 1 has been rewritten to better reflect the structure of all previous chapters. The code fault injection testing has been completed by modifying the code in the OpenAQ library folder containing "measurement-schema.json", which contains all the requirements for reading in sensor data, and then reporting the results of how our test cases were affected. Lastly, our previously written HTML report was updated with test clarifications to make the overall understanding of our tests more comprehensive to everyone.

Learning Outcomes

Our team has gained further understanding of how important a comprehensive document is. Our team has improved on writing code in HTML, and BASH. The work done helped develop a further understanding of how to design and implement a comprehensive report without breaking the project. Continued proper usage of draw.io (online) was made familiar through implementing the Activity Diagram for this deliverable. The fault injection taught our team how easy it is to make a complex software not work properly, and also how even the best programs do not have the best fault tolerances.

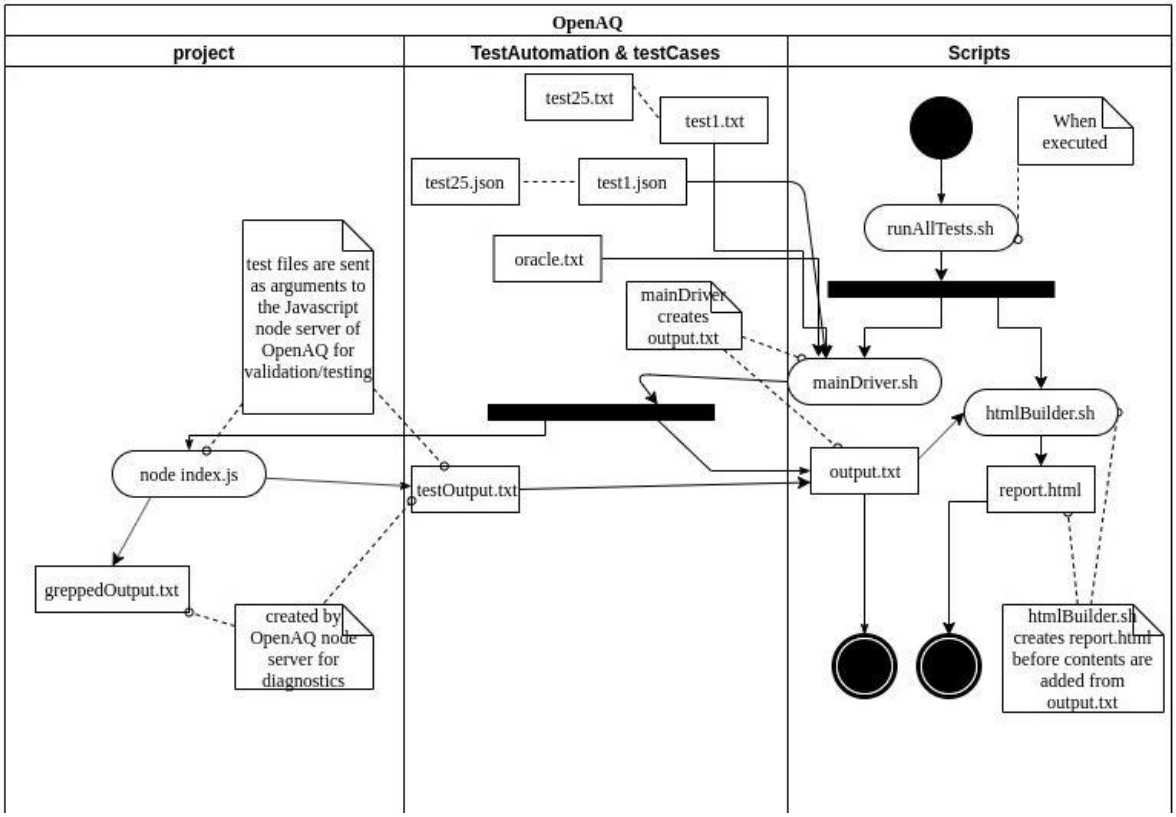
Problems Experienced

During a reiteration of the HTML code for added clarification of our report of the tests, many of our tests failed. The solution around this problem was to add clarification within our test case files instead of within the HTML report.

Going forward

The next goal, per specification of our test plan, is to finalize our automated testing framework for any documentation that may be missing. Finish the project poster and presentation for our "Final Presentation Day". Our group also needs to write our overall learning experiences for this whole project for Chapter 6 in the final report. We also need to review all of our previous chapters for accuracy and comprehensiveness.

Activity Diagram for our project can be found below, as well as a further detailed report of our code fault injection tests.



Below is our code fault injection analysis of our project.

Inside the TestAutomation/project/lib folder is the measurement-schema.json file which contains all the required fields needed for a sensor's data. We'll be creating faults in the code in this file and attempt to predict the effects on testing.

1. Changing minimum latitude allowed from -90 to 0. Done by changing Ln 57 of measurement-schema.json from: "minimum": -90, to: "minimum": 0,

Expectation: This should cause all test cases with a negative latitude to fail.

Before: Latitude minimum: -90

16	test16	Latitude can only range from 0 to +-90 degrees	{"parameter": "pm25", "unit": "ppm", "value": 14, "date": {"local": "2016-01-24T19:00:04+00:00"}, "coordinates": {"latitude": -100, "longitude": 34}}	instance.coordinates.latitude must have a minimum value of -90	passed
17	test17	Latitude can only range from 0 to +-90 degrees	{"parameter": "pm25", "unit": "ppm", "value": 14, "date": {"local": "2016-01-24T19:00:04+00:00"}, "coordinates": {"latitude": -91, "longitude": 190}}	instance.coordinates.latitude must have a minimum value of -90=1, instance.coordinates.longitude must have a maximum value of 180=1	passed

After: Latitude minimum: 0

16	test16	Latitude can only range from 0 to +-90 degrees	{"parameter": "pm25", "unit": "ppm", "value": 14, "date": {"local": "2016-01-24T19:00:04+00:00"}, "coordinates": {"latitude": -100, "longitude": 34}}	instance.coordinates.latitude must have a minimum value of -90	failed
17	test17	Latitude can only range from 0 to +-90 degrees	{"parameter": "pm25", "unit": "ppm", "value": 14, "date": {"local": "2016-01-24T19:00:04+00:00"}, "coordinates": {"latitude": -91, "longitude": 190}}	instance.coordinates.latitude must have a minimum value of -90=1, instance.coordinates.longitude must have a maximum value of 180=1	failed

Results: Only test cases that were made for latitude testing were affected. Notably test 16 and 17, which failed since the oracle is now incorrect. The oracle for these two was looking for the error message "instance.coordinates.latitude must have a minimum value of -90", but the program now throws "instance.coordinates.latitude must have a minimum value of 0".

2. Changing latitude maximum to 0. Expecting it to cause failures for tests that were testing the upper bound of the latitude field. Done by changing Ln 58 of measurement-schema.json from : "maximum": 90 to : "maximum": 0

Expectation: This should affect any test cases that dealt with latitude, causing them to fail.

Results: No test cases were affected. This is due to not having any test cases with oracles that search for latitude errors.

3. Changing longitude minimum to 0. This is done by changing Ln 62 of measurement-schema.json from: "minimum": -180 to: "minimum": 0

Expectation: This this make any test cases with longitude inputs that are < 0.

Results: Test 20 went from passing to failing, since it was expecting the longitude minimum to be -180, not 0. Other tests were unaffected. Even ones with negative longitude still passed, since the oracle each test case is looking for is still found in the report.

4. Changing maximum longitude to 0. This is done by changing Ln 63 of measurement-schema.json from: "maximum": 180 to: "maximum": 0

Expectation: Similarly to other tests, any test case whose oracle is looking for a longitude high out of range will fail. The rest should still pass.

Results: Tests 15, 17, 18, 19, 21, and 22 failed. All but 21 and 22 were based on the longitude parameter. 21 and 22 failed because they both contain positive longitudes, and the oracle for each expects no error messages.

5. Adding "pm20", a phony parameter, to the lists of accepted parameters. This is done by changing Ln 12 in measurement-schema.json from: "enum": ["pm25", "pm10", "no2",

"so2", "o3", "co", "bc"] to: "enum": ["pm25", "pm10", "no2", "so2", "o3", "co", "bc", "pm20"]

Expectation: This should only affect test case 9, which should now fail. Test case 9 is expecting an error due to using the phony "pm20" as a parameter.

Results: Test case 9 is the only test case that failed. Just as expected!

Interpretation of overall results: A lot of the code faults did not affect many test cases. This is due to how the test cases are written. They are looking for the specific error that matches their oracle, which will show up in the report along with any new errors, so they still pass. With more specific oracles, the test results would be more likely to change from a pass to a fail.

Chapter 6

Overall Experiences

The project has been very insightful for what software development within a large project may include. The success of a complex software is very dependent on having a strategy to manage the workflow of the project. Communication within a group project is very important as well to make sure everyone is on the same level of understanding the tasks to complete within a huge project. Divide and conquer of project work is not always the method of working as a team, collaboration is essential to success. Our team has learned a decent understanding of writing code in BASH, and HTML. A basic understanding of JSON was gained to incorporate the test case files within our particular project. Our team has become very comfortable with running a Linux environment while working with our semester-long project. The deeper understanding of Test-Driven Development was established through the phases we iterated through such as developing a test plan to follow. Following the test plan throughout the duration of the project helped establish a clear sense of direction and time management for our group, individually and as a team. We learned how to use draw.io to produce our diagrams pertaining to our project implementation's "Layered Architecture" and "Activity Diagram". Through the beginning of our frustrations with the project we learned just how important comprehensive documentation is when running somebody else's code. And lastly, the fault injection portion of the project taught our team how easy it is to make a complex software not work properly. Reinforcing the importance of fault tolerance within a project, which can only be established through testing.

Self-Evaluation of Team's Work

Throughout the project, our group would meet every other Monday consistently in order to stay on top of our due dates according to our test plan. Every assignment was always given everyone's full attention until complete. If there were any issues completing a particular assignment, assistance was pursued amongst the other groupmates to resolve. If nobody else could resolve the problem then feedback would wait to be received by the instructor during each deliverable when we presented any issues we were facing. Overall our worked diligently and always communicated whether face to face or through social media applications such as Discord.

Project Suggestions

The only project assignment suggestion for next semester would be to include further instruction on the requirements of the scripting portions of the project. State the goals clearly and maybe some direction on how they can be pursued.