OpenAQ

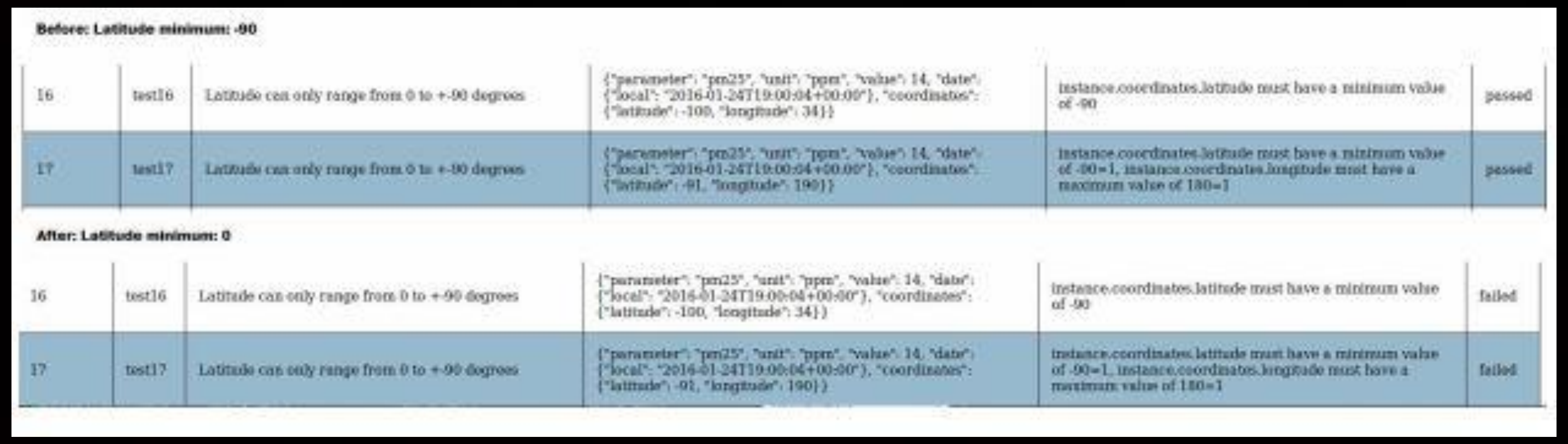
Matthew Anuszkiewicz, Benjamin Duke, Tino Pimentel
CSCI 362: Software Engineering

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. Later we changed the source code within OpenAQ-Fetch to see if our set of tests would still execute correctly, known as fault injection testing.

How Everything Works

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. Once the results are finished being collected, they are then submitted into a text file that our HTML program will use to build a comprehensive report for people to understand if any particular test case passed or failed.



Before: Latitude minimum: -90

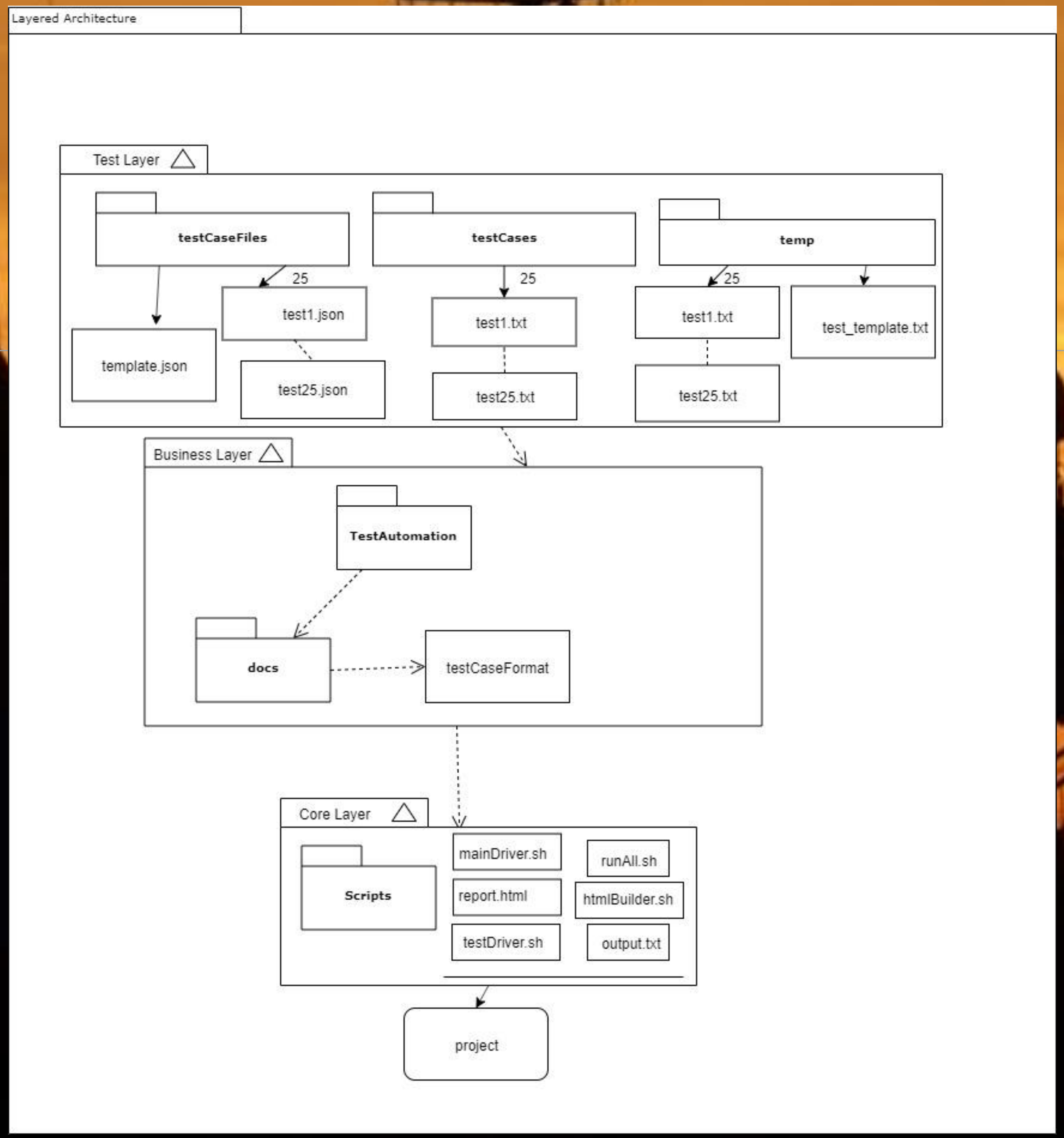
16	test16	Latitude can only range from 0 to +-90 degrees	{"parameter": "pm25", "unit": "ppm", "value": 14, "date": {"local": "2016-01-24T19:00:04+00:00"}, "coordinates": {"latitude": -100, "longitude": 34}}	instance.coordinates.latitude must have a minimum value of -90	passed
17	test17	Latitude can only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

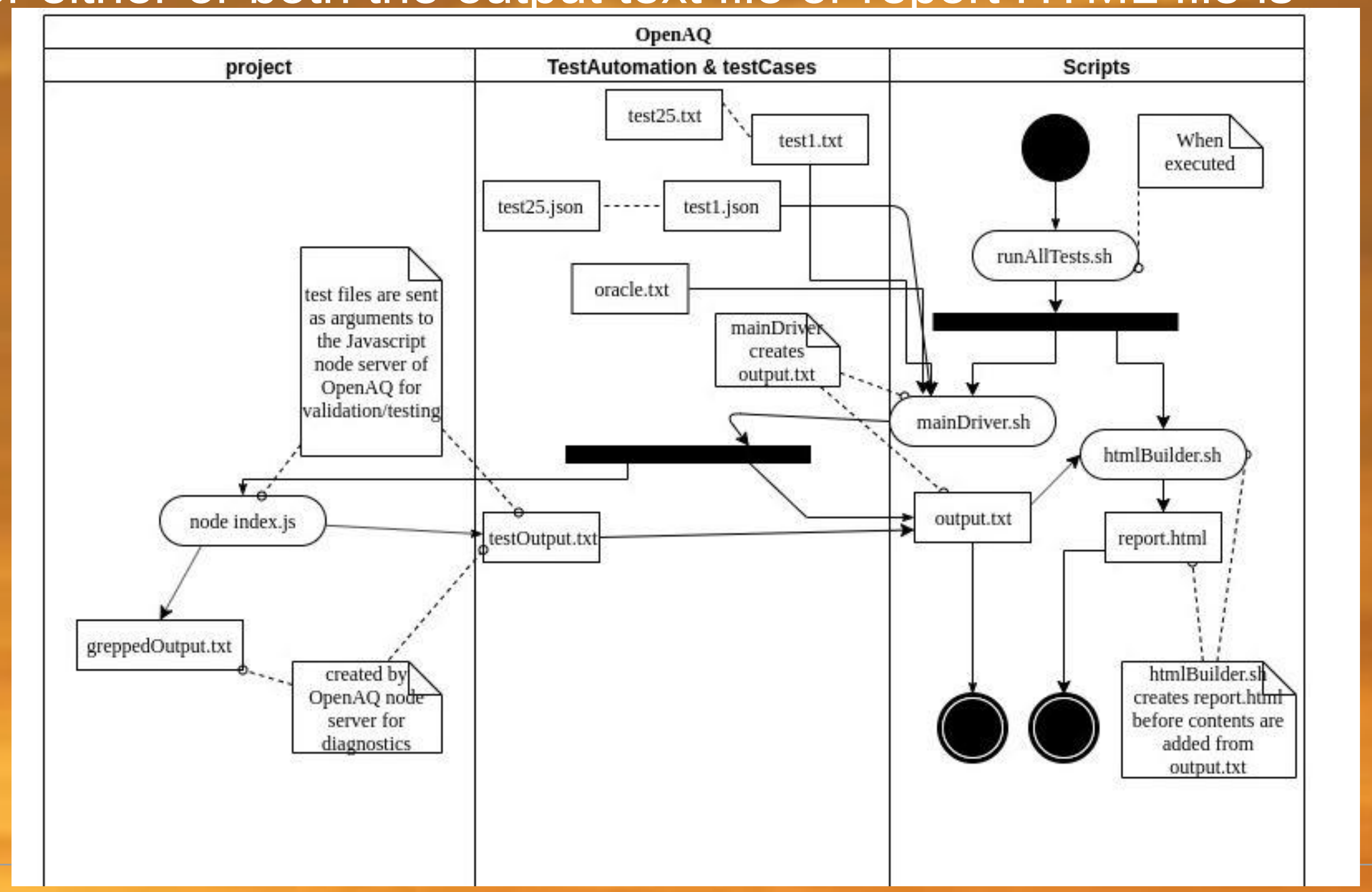
Architectural Breakdown of Framework

Seen below is the package architecture of our automated testing framework. The architectural pattern depicted here is referred to as a "Layered Architecture", where there consists of a Test Layer, Business Layer, and Core Layer. The arrows indicate a transmission of data or direction of data. The Test Layer consists of our actual test folders which hold our test case data to be passed into the OpenAQ project. The Business Layer consists of the documentation needed for any other developer to read and understand how to write a test case themselves. Lastly the Core Layer contains our Scripts folder which holds the main automated testing framework that executes instructions to OpenAQ-Fetch's HTTP server using input from the Test Layer. Every layer builds up to executing instructions into the project folder where we keep OpenAQ-Fetch.



Activity Diagram of Framework

Seen below is a UML (Unified Modeling Language) diagram referred to as an "Activity Diagram". The arrows indicate direction and message flow, or data flow. As time progresses downward within the diagram, the different parts of the automated testing framework execute instructions. The framework begins with the command "runAllTests.sh" to begin the execution of the testing framework where an HTML report will be built using the feedback from an output text file. The framework has finished running after either or both the output text file or report HTML file is produced.


OpenAQ
project
TestAutomation & testCases
Scripts
test25.txt
test1.txt
test25.json
test1.json
When executed
runAllTests.sh
oracle.txt
test files are sent as arguments to the Javascript node server of OpenAQ for validation/testing
mainDriver creates output.txt
mainDriver.sh
htmlBuilder.sh
node index.js
testOutput.txt
output.txt
report.html
greppedOutput.txt
created by OpenAQ node server for diagnostics
htmlBuilder.sh creates report.html before contents are added from output.txt

Lessons Learned

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.