Testing Document

Students:

Brian Quilty - 18373856

Conor Marsh - 19728351

Supervisor:

Paul Clarke

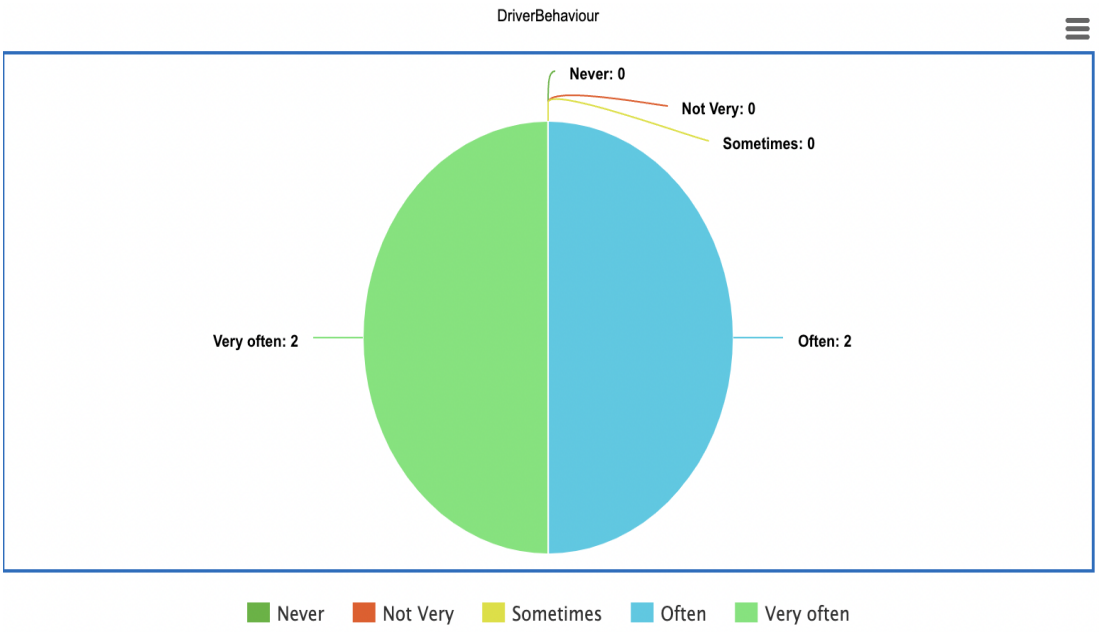
06/05/23

10 pages long

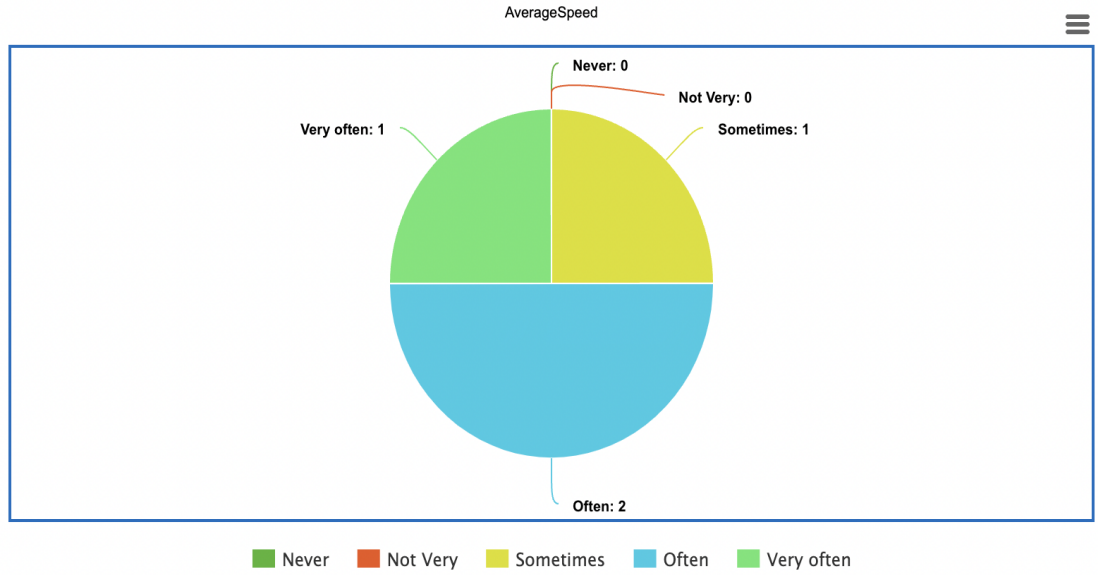
1. User Testing

We selected 4 people to partake in our user testing. To get feedback on the app's functionality we asked a series of questions to be able to gather users' opinion and feedback of our app.

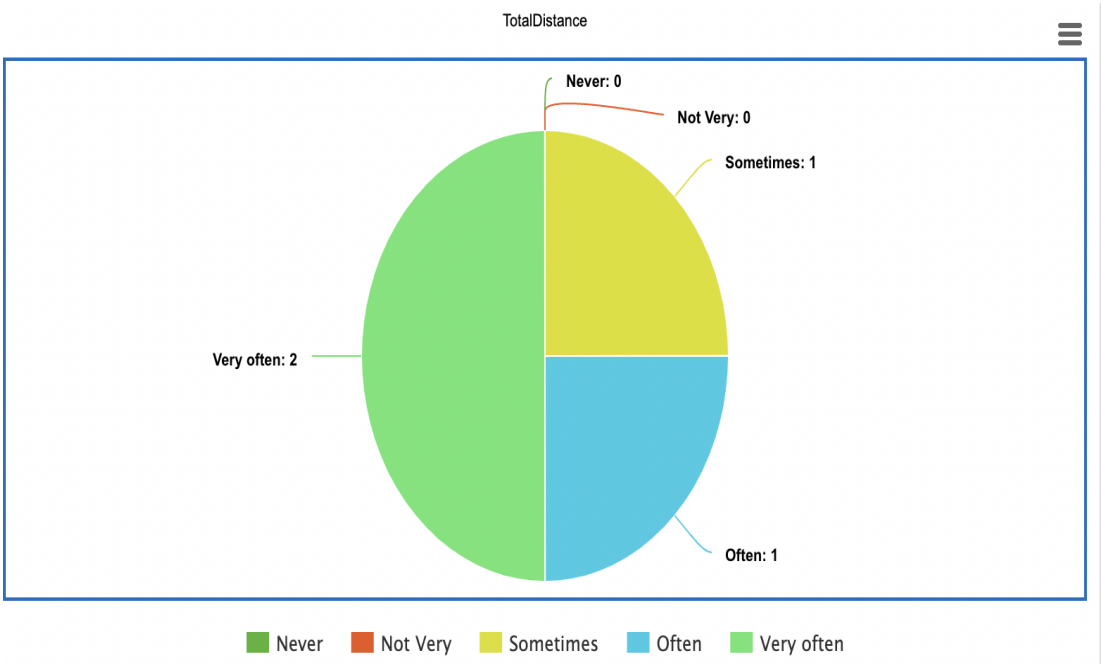
First Q : How accurate did you feel the app was at predicting the driver's behavior?



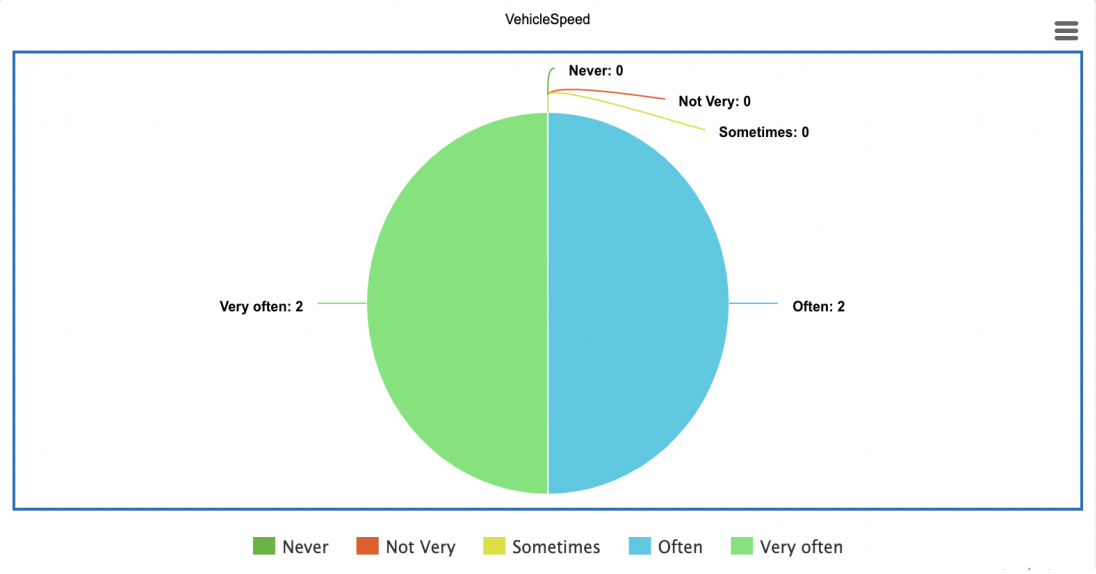
Second Q : How accurate did you feel the app was at obtaining the average speed?



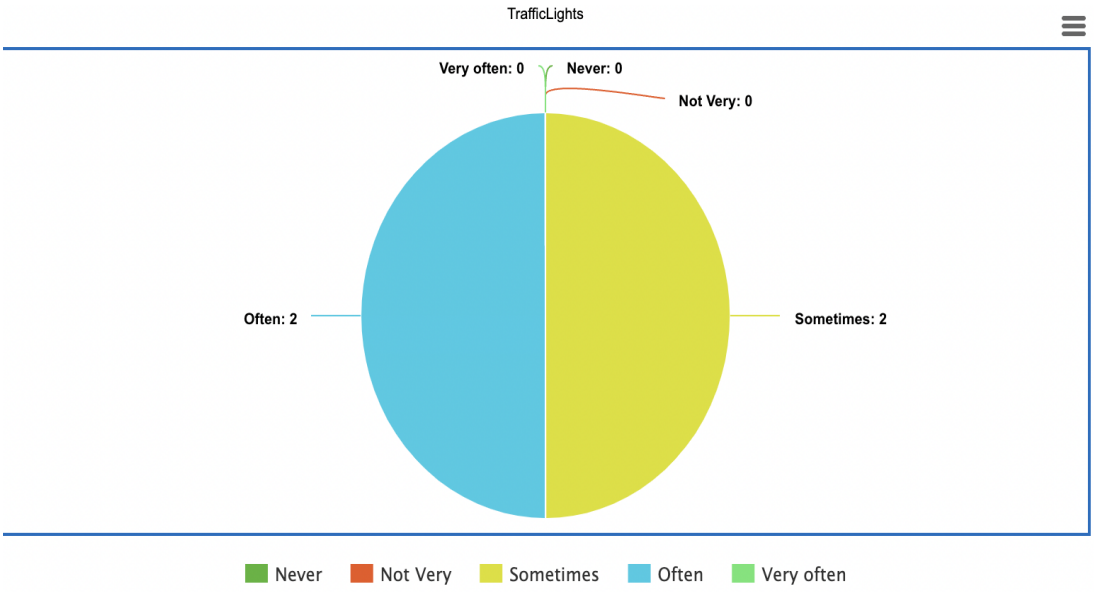
Third Q : How accurate did you feel the app was at obtaining the total distance of the journey?



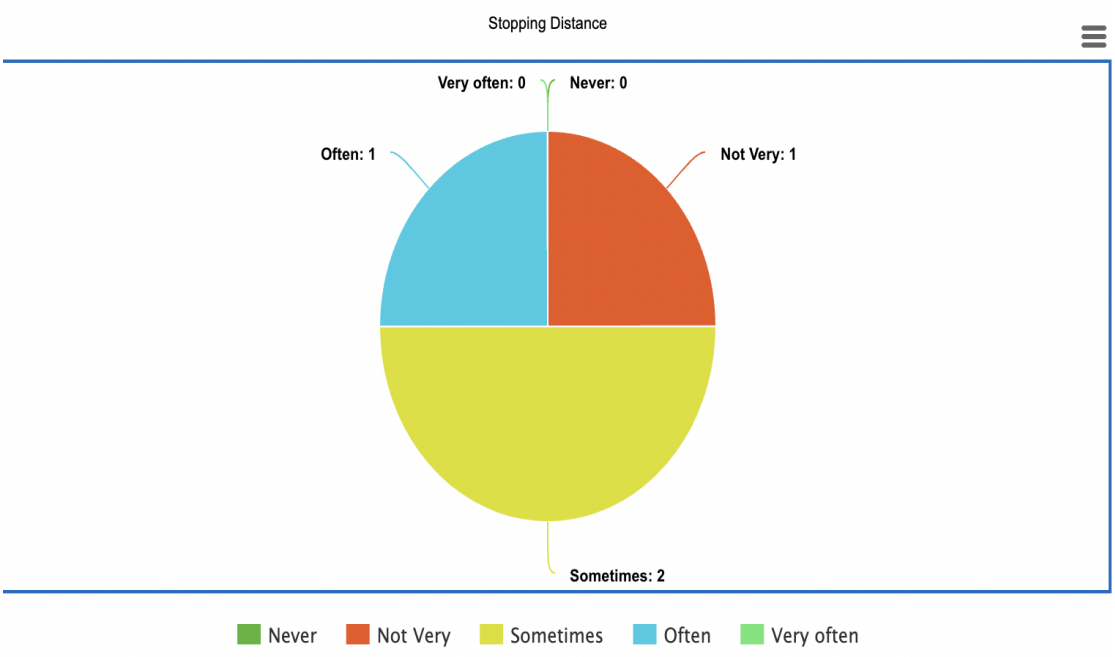
Fourth Q : How accurate did you feel the speed of the vehicle was?



Fifth Q : How accurate did you feel the app was at predicting traffic lights?



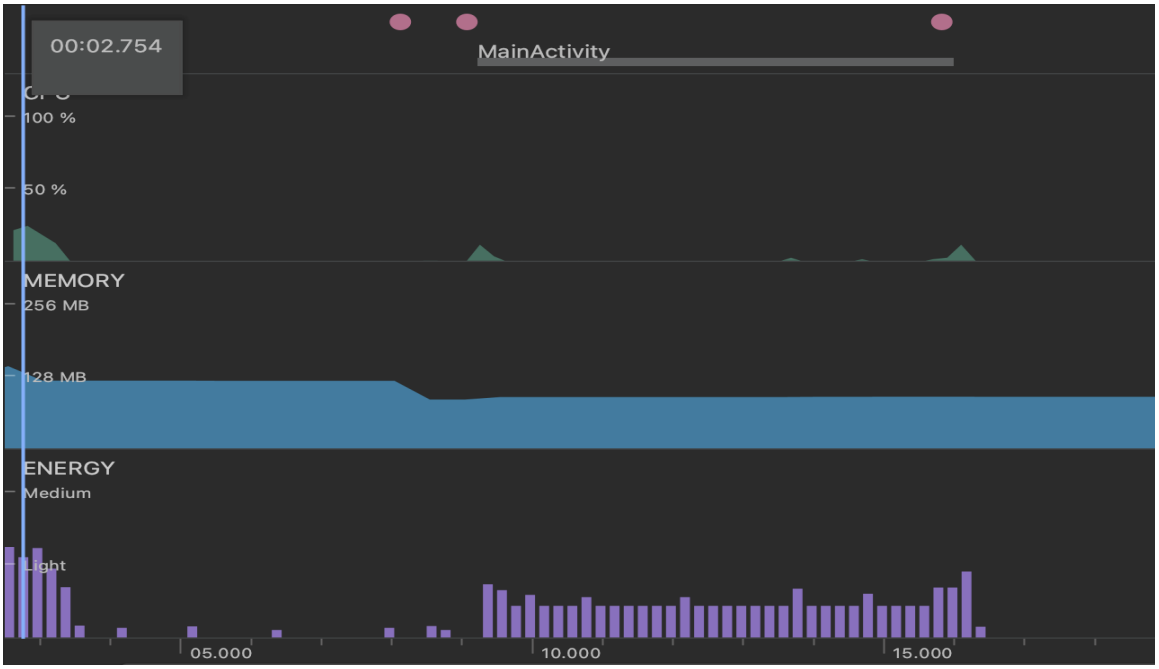
Sixth Q : How accurate did you feel the app was at predicting car stopping distances?



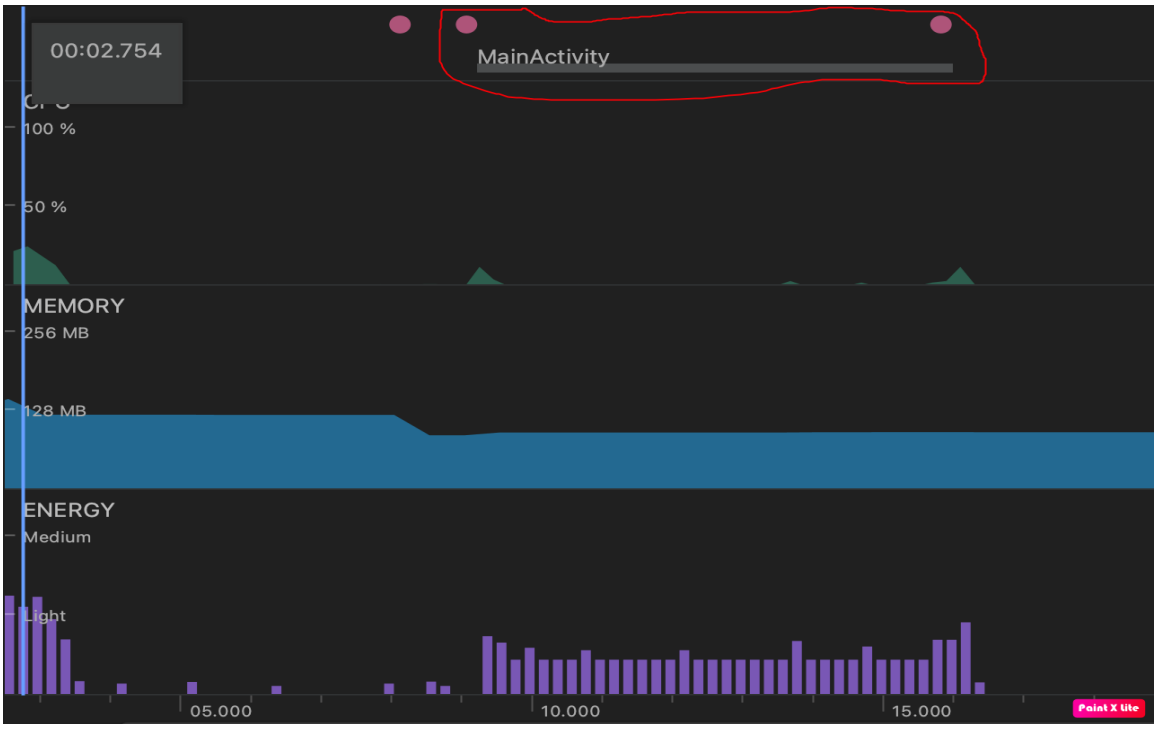
2. System Testing

To conduct system testing within our app we used android studio's profiler to get various statistics while running the app such as cpu usage and memory usage.

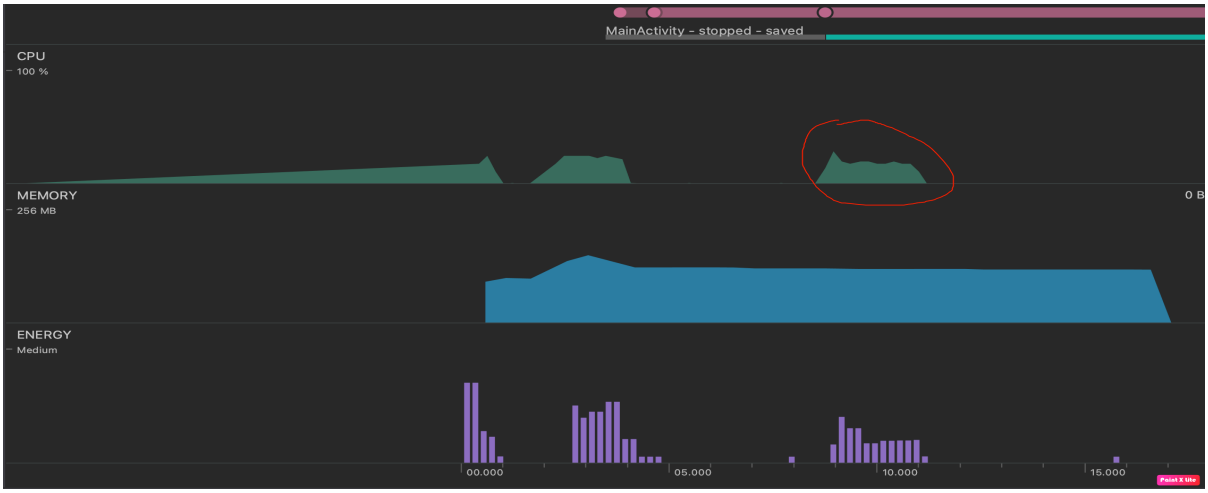
We first used the profiler to test when the journey is ongoing.



This is the profiling of the system when the journey has started and ended.

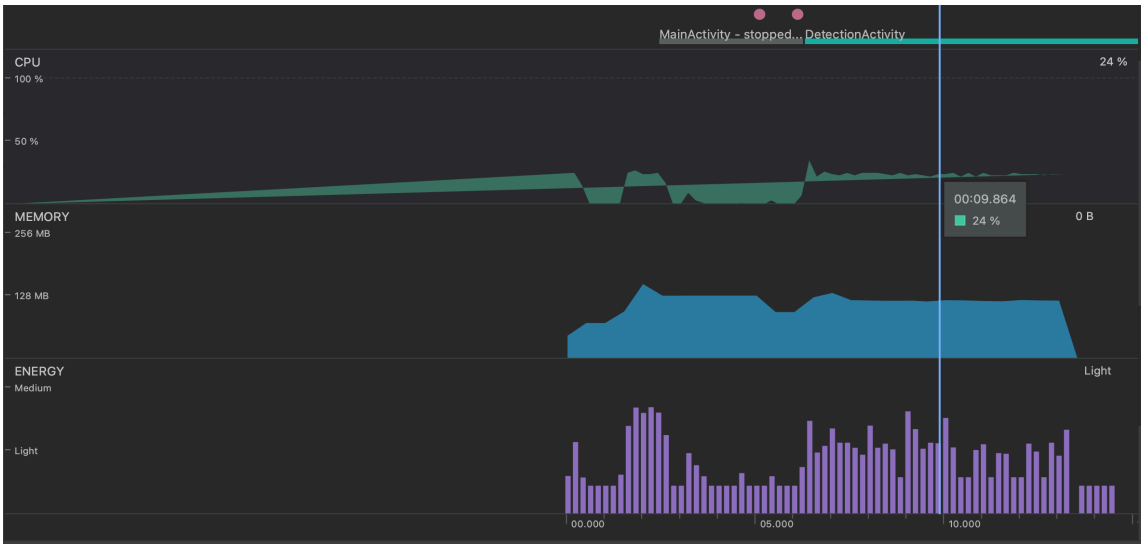


The highlighted red section is the journey starting and ending. As you can see it uses a light amount of energy throughout the journey, which is good. It also only uses about 20% of CPU memory when the journey starts and finishes. In between the journey it uses barely any CPU. This means that running a journey is not very intensive for the phone, making the journey functionality relatively lightweight.



The spikes in CPU notate when the app started and when the app changed screen. The rest of the CPU usage was low as the journeyDetailsScreen is a static screen that only displays information. For every screen switch we can assume that the CPU uses around 20% consumption. This is relatively small and not a massive spike that would raise too many problems.

This is the profiling for the individual journeys activity. Again, upon opening the screen, the CPU spikes to around the 20-25% mark. The two spikes, one showing 17 and the other around the same were from clicking into a journey, we also deleted a journey which didn't cause a spike in CPU. Clicking into a journey is like clicking into a new screen so we know why there was a spike in CPU percentage. There is nothing here that is really too concerning.



This is the profile for the detection activity screen. As you can see it uses relatively more energy than the other sections of the application and also uses a lot more CPU than the other sections of the app. The other sections of the app spike at around 20-25% when changing screens but otherwise remain fairly stagnant for CPU usage. In this screen however, the CPU usage remains at a steady rate of 20-25%. This is because the phone is constantly having to detect objects which takes up a relatively large chunk of CPU resources. The energy usage is also a lot more as it peaks in the medium levels of energy usage. This would mean that the detection section of the app would need a relatively good phone to be able to use this section of the app. It does show that it does not take up an absurd number of resources and is very much useful for medium to high range android devices. This could be something that we could note in the advertising section if we were to release this app.

3. Unit Testing

For our unit testing we created tests for each java class testing various functionality for our app. To test the classes, we used the Robo Electric Runner class. These tests can be found in our app under the subfolder -> app -> src -> test -> java -> com -> example -> autotrackerca400. These tests were automatically run whenever a commit was pushed to GitLab as we had setup CI/CD hooks, which would automatically run all the tests in this folder using the command, “./gradlew test”. You can run the tests in android studio by hovering over the folder and right clicking and then pressing run tests in com.example.autotrackerca400.

4. Integration Testing

Our integration tests were created to test the functionality of the UI. It uses the Junit class to simulate a user using the app. The tests consist of clicking and selecting different UI aspects within the app. The tests can be found in our app under the subfolder -> app -> src -> androidTest -> com -> example -> autotrackerca400. We unfortunately couldn’t set up git hooks to execute these tests when pushing to GitLab as the emulator on GitLab could only be used with Api 24, which would cause the tests to fail. You can run the tests in android studio by hovering over the folder and right clicking and then pressing run tests in com.example.autotrackerca400. If you can't get the integration tests to run inside android studio, then -> cd to the android studio project folder on the command line. Make sure your android studio is open and that you have an emulator installed, then run the command ‘./gradlew connectedAndroidTest’. Also, there are some mainactivity integration tests that need you to manually allow location services. Unfortunately, this can't be hardcoded only prompted so you will have to look out for it in the emulator in android studio, or else some tests will fail because of it.

5. Adhoc Testing

We implemented adhoc testing throughout our development process. Adhoc testing means that our testing had no real planning behind it and was more off the cuff compared to our unit and integration testing. Whenever we implemented

new features, we used methods like java's system.out.println method to ensure that the functionality produced the correct output. If we ever encountered a bug and were unsure of what was going wrong, we would place print statements in lines around where the functionality was implemented to see what was going on and going wrong. For example, for ensuring that the speedlimit was correct we printed out the value originally instead of having to assign it to a text inside the mainactivity, this saved time and ensured we were getting the correct value before having to implement the UI functionality to show the value.