Unit Testing Analysis

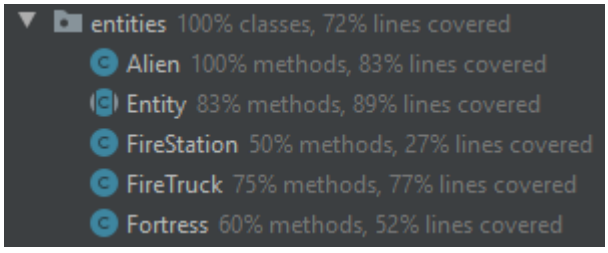
This small report will breakdown and analyse the testing results and coverage. It will be covering the line percentage coverage, method percentage coverage and time efficiency.

First it is important to mention that almost all tests conducted in this part of the project were Unit tests and not user-tests. With the large extent that DicyCat user-tested their product, and with how little they had unit tested it, we felt it was best to spend most of our time ensuring the code we inherited had no flaws.

Line Percentage Coverage:

As mentioned in our previous report, line percentage coverage should not be the lone indicator of how strong the unit test suite is. It is, however, a common starting point for analysis, as it will be in this report.

Intellij offers a feature that will run all tests and calculate line percentage coverage. It will break this down from the whole project to individual classes which provides a wealth of information. When we run our tests with this feature and for example look at the 'entities' folder, which hosts core classes to game play, we see this:



We see that the line coverage for the file containing all 'entity' like classes is 72%. With a high of 89% in 'Entity' and a low of 27% in 'FireStation'. This essentially tells us how many lines our tests passed through out of the total lines of each class. However, if we take a closer look at 'FireStation' we can see that methods inside are for visual purposes or have been user tested. On the other hand with 'Entity' almost all lines are covered and those not, for example 'update()' do not accomplish anything, thus there was no need to send time testing. There is the shortcoming of line percentage coverage, it cannot discrepant between the functional lines and the pedantic lines.

Method Percentage Coverage:

Similarly to the section on line percentage coverage, method percentage coverage is not an all encompassing metric but it is helpful. It will see the amount of methods tests go through compared to the total amount. It has the same flaws as those in line percentage coverage. If we again look at the 'entities' example, we can see that FireStation has a decent method coverage but a poor line coverage. This discrepancy indicates that while not many lines of code were covered, they were essential to ensuring the requirements were tested.

Time Efficiency:

This analysis is very strong and often independent of other coverage metrics. The general indicator that this can give is not the strength of the test suite but its quality. It is key to software development that tests are fast as well as strong, since the developers should not need to long to see if what they have written works as needed and intended. Thus the lighter the code, the faster it is, the more efficient it is, and the better the overall testing suite is.

Gradle keeps all our timing for the tests that we run, which we have exported into two files, the individual test times and the overall time. The overall time we achieved is 4.369s, which is good, since with 44 tests the average time per test is around 0.09s. However looking at individual tests we find that there are a few tests that take about 300ms, which is slower than we want. We found that this was due to initialization of classes within tests and not at the @Before section. In our next iteration of testing we will pay closer attention to the time of individual tests to ensure this does not happen again.

Summary:

Overall, we feel that this iteration of testing was an important lesson on how to take over another team's project. It is both key to ensure a strong base line of testing that we had received but also to start the retesting with a better indication of overall project structure. The testing suite is definitely fast and covers critical areas of code, such as entities, bullets and gameObject, to a high degree. It was not as well covered in the area of the miniGame, which relied solely on user testing or scenes which we do not think are functionally critical.