Software Testing Report

Our initial approach to testing was to try to automatically test as much as possible in our code. We decided to use JUnit as our testing framework. Using this framework will allow us to create tests in an easily executable way. To prevent brittleness in our tests, which is useful as we can expect frequent changes to the code being tested, we used public interfaces to build our JUnit tests, as well as testing states rather than interactions (as these are likely to change). We also created tests that are concise, so it is easy to work out what is being tested, as well as making error and fail output strings that make it clear which test failed.

A further approach we used to create effective tests in JUnit was DAMP tests, giving each test a clear name, so when viewing the tests as a whole, it is easy to see what is being tested and where the coverage is lacking. We realised that aiming for 100% cover is an arbitrary goal that is not necessarily achievable. It was a much clearer goal to look once we have made tests and see if we have missed any important sections. Using DAMP tests also made it clearer what was being tested at first glance, and when someone was reviewing the tests, they wouldn't have to search through all of the methods to understand what is being tested.

When we first started to develop tests, we considered using test doubles. Test doubles allow a component to be tested without having to implement other components. However, the codebase we took on would have required significant refactoring. It could also have overcomplicated tests that would have been simple to create without using test doubles. Therefore, we decided not to use test doubles.

Due to difficulties with running a successful Headless Client, a lot of the testing has been implemented manually. We achieved this by adding printline statements into certain methods, so we could log in the console when they were called. This approach was used when testing the scoring system, particularly adding buildings to the network and calculating distance related bonuses.

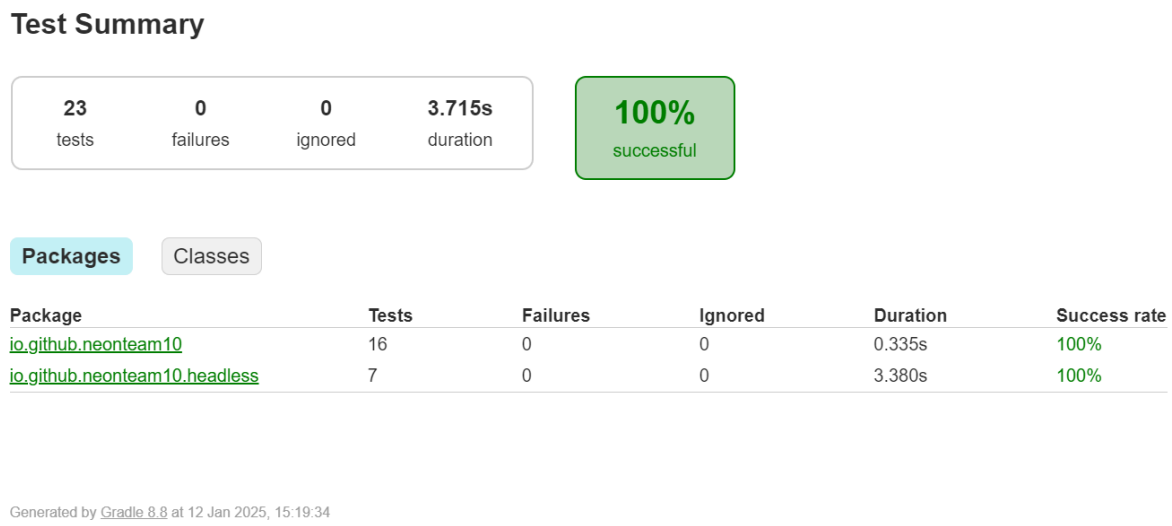
After much work, we succeeded in creating a Headless application of our game. However, since so many of the classes are co-dependent, it's inefficient to create the Headless Application for every test, and so a decision was made to focus on play testing for features such as building placement, and the score and events display.

When creating tests for the game, we looked at the requirements. To ensure that the game works as it should, the majority of the requirements elicited need to be tested, otherwise significant sections of the game will not work properly. To make this easier to do we aimed for as many tests as possible to be automated in JUnit, however, we knew that some tests would have to be manually carried out, and that these would need to be documented with identifiers, steps, outcomes and what the category is.

Our project uses a graph object as a method to determine where buildings can be placed, as well as where any buildings are actively being placed. One of our tests, BuildingGraphTest, attempts to place a building on the graph. Between doing this it returns graph data both before and after placing a building on the graph, which allows us to identify whether placing the building was successful through an updated graph.

Report on Junit tests

Our Junit test coverage comprises 23 tests, of which none are failing under the current implementation.



Test Summary

23	0	0	3.715s	100%
tests	failures	ignored	duration	successful

Packages Classes

Package	Tests	Failures	Ignored	Duration	Success rate
io.github.neonteam10	16	0	0	0.335s	100%
io.github.neonteam10.headless	7	0	0	3.380s	100%

Generated by Gradle 8.8 at 12 Jan 2025, 15:19:34

Whilst this by no means assures the validity and robustness of our code, it offers some confidence in the building system we have created.
Our tests do not test libgdx methods and gameplay, as we are assuming that all functionality we expect will work due to it being a very popular and widely used library, and so will employ within itself strict testing measures.
Instead, our tests focus on the unique aspect of our implementation, namely, the graphs package we created specifically. The tests check the creation of buildings, and the adding of buildings to the graph.
Our tests are split into two packages - one which has absolutely no dependence on any libgdx or game ui, and the other which, in order to run, would require a headless application. The standalone tests create instances of the building without any data or type, and as such are only useful to a limited extent.
Due to difficulties running the headless client, our tests doing this with building prefabs are incomplete, and so we have had to resort to manual testing for these aspects.

Our manual test plan can be found here, and outlines the test cases written in BDD that we have identified as procedures of importance. As you can see, it aims to cover all the different workflows that can arise from each specific state.
Our manual tests were created by considering the requirements and the user's gameplay, as well as filling in any gaps missed by the autotest. As a result, there is a clear link between our tests, and the system and user requirements.
When testing the 'Accommodation building distance bonuses' and 'Canteen building distance bonuses', we added code to log to the terminal when the bonus changed, so we could place the buildings and verify the bonus was correctly calculated and called. We realise that this was a perfect opportunity for unit testing, but due to unforeseen difficulties, we were unable to and so were forced to manually test each scenario.

Full reports can be found here:

- https://neon-team-10.github.io/assets/test/index.html
- https://neon-team-10.github.io/assets/jacocoHtml/index.html