# Cohort 3 - Group 22: The Wafflers

## Eng1 Assessment 2

## Software Testing Report

### Team Members:


Tunahan Sisman
Yousef Omar
Merrill Davis
Tom Comrie
Rohan Sandhu
Cameron Pounder
Daniel Redshaw

4a) Tests were written for either validation, whether the game fits the customers requirements, or  for verification, whether the game fits the specification.

It was aimed to have as many of the tests be automated as possible, this was because it allows bugs to be caught much sooner, as they can be run at every change, and allow changes to be released much more quickly.

A traceability matrix was made which linked customer requirements to the tests written, used to identify which tests were not covered by automated testing and required manual testing.

This was appropriate as it is difficult to achieve 100% test coverage, and additionally difficulties were encountered with testing areas of the game utilising libraries from LibGDX.

For this reason, ensuring that code coverage for the core folder, which contains all the logic functionality of the game, was as close to 100% as possible was the priority. This was because manual testing could then be done to check the GUI layer worked and automated tests could ensure the backend functionality works.

When creating automated tests, the input space partitions and their boundaries were considered, which determined the inputs to ensure maximum coverage. Tests were written in a DAMP format, so that tests are self-evident, as recommended by the “Software Engineering at Google” book[1].

Test doubles, using mockito, were used to act as fakes and stubs to remove dependencies in tests to test parts of software in isolation, this was done to make it easier to identify where errors are caused in code, compared to larger scale tests such as integration tests which involve multiple classes or methods.

Exploratory testing was undertaken using the testing matrix (linked at the bottom of the page) and the coverage tracker built with IntelliJ IDE and jacoco test reports, which highlights lines and branches which are not covered. This allowed the team to identify possible areas to automate tests with.

When automation was not possible or desirable, manual tests were created to ensure the project met the requirements. These tests can be viewed in the manual testing guide on the website, and linked at the end of this document.

Where possible, within resource constraints, code was refactored to improve testability, particularly if it could be moved outside of GUI related methods it was then possible to add automated tests.

4b) The overall line coverage was 30% and branch coverage was 47%. The areas which were not covered by automated tests were predominantly classes and methods that involved GUI elements, such as textures and tilemaps. All 70 automated tests pass, however of the 9 manual tests, the test checking the music and sound effects fail due to both not being implemented, as set out in the implementation deliverable, with all other manual tests which are non-subjective passing manual tests.

For the two manual tests testing the music to pass the music and sound effects would need to  be implemented in the game.

As mentioned above, the team decided that it was most important for the automated tests to cover the backend, leaving GUI components to be tested manually, due to the difficulty in testing them.

The main folder for the backend, labelled core, had 99% line coverage, and 96% branch coverage. The missing coverage can be explained due to switch statements where due to typesetting, it is very difficult to have an input which would result in the default output, for example with the Exitconditions class, due to overriding the *tostring* method, or in the core class, where all previous activity types are accounted for due to if statements or switch cases. It was ultimately decided that it was not worth the resources to refactor the code to avoid having catching statements at the end to have a superficial increase in coverage.

For clarity, key backend classes are: Core, Time, Energy, Leaderboard, ActivityLocation, and numerous enumerator classes.

The backend tests achieve a high level of functional suitability as set out in the ISO 25010 standards[2]. They have high levels of functional completeness, covering all logic related components of the game in relation to the requirements. This is evidenced by the testing traceability matrix, with the majority of functional and user requirements being met by at least one automated test, with requirements listed as covered by both manual and automated test cases being partially covered by automated testing.

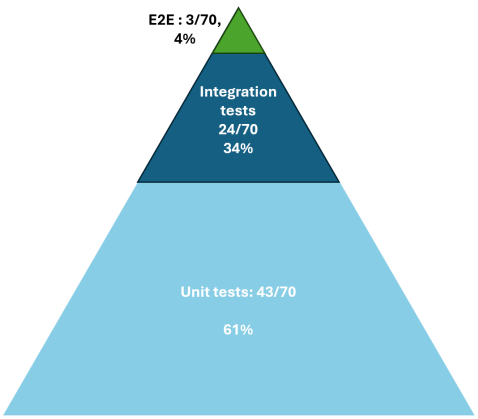
This typically is due to the backend being tested but a manual test required to check the GUI layer is working, for instance with the interaction popups when near an activity, the method checking whether an interaction is nearby is tested, and the assets used to create the popup are tested to exist, however a manual test is required to check screen class uses these correctly to present the popup. Another example is that the methods inside the core class that handle interaction is tested fully, however a manual test is required to check that the playscreen is correctly calling this method, which is validated by doing the activities and checking whether the counters in the bottom right change, as their output is validated by the backend testing, as set out in the manual testing guide linked at the bottom of this document.

As evidenced by the high branch coverage, the backend tests have great functional correctness. Care was taken to ensure that the tests also have a functional appropriateness, “degree to which the functions facilitate the accomplishment of specified tasks and objectives” [1]. For example, in the scoring tests instead of taking a certain input of a mix of activities performed over the week, and verifying that a particular score is given out, the tests instead validate that the scores produced meet the requirements set out by the customer. In

particular, the tests would check whether studying more resulted in a higher score, assuming other inputs are the same, but does not specify exactly how much greater the score should be. Similarly this is done for studying too much, along with the requirements set out for how eating regularly and recreation should affect the score.

This ultimately has resulted in more robust tests that continue to validate correctly whether or not the scoring system fits the requirements, without having to rewrite the tests each time the scoring algorithm is changed.

The distribution of the automated tests was as follows:




The “Software Engineering at Google” textbook recommends a 5,15,80 split between E2E, integration and unit tests.[3]

The greater than suggested number of integration tests in this project was due to the interdependency of certain classes, and the use of the GDX headless backend to run asset tests, as well as tests on the playscreen and avatar class.

The mockito mocking framework was used to reduce the number of tests that became too large due to these dependencies by mocking the return of a function, or its changes to internal states. Particularly in large classes like Core, mockito spies were used, which allows for mocking of the outputs and changes to internal states from other methods in the Core class, not to be confused with the core folder.

For the user interface layer, the automated tests which were performed include checking that there is the correct number of activity locations as set out in the requirements, and that each of them is within the map area. There are also asset tests that check the files used by the screen and other GUI related classes.

For the most part the remainder of the graphical interface is tested manually, such as checking buttons to set the correct new screen, that when a location is interacted with it correctly calls the appropriate backend methods, and that the animations created from the sprite sheet assets are correct.

The relationship between manual tests and the requirements can be seen in the traceability matrix, with manual tests listed at the bottom of the requirement identifier column, on pages 3,4,6 and 8 if reading the PDF version of the matrix.

**Testing results and coverage report:**
https://uoycs.github.io/HeslingtonHustleAss2Website/A2/testing/index.html

**Manual testing guide:**

**Testing Traceability matrix:**

If there are any issues with these links please find the appropriate content at the website at:
https://uoycs.github.io/HeslingtonHustleAss2Website/

**References:**

[1] Winters, Titus, Manshreck, Tom, and Wright, Hyrum. Software Engineering at Google : Lessons Learned from Programming over Time. (P248)

[2] ISO-25010 software product quality [online] available:
https://iso25000.com/index.php/en/iso-25000-standards/iso-25010

[3] Winters, Titus, Manshreck, Tom, and Wright, Hyrum. Software Engineering at Google : Lessons Learned from Programming over Time. (P220)