Testing

C3 Group 6
Team WHNI

Aaratrika Das
Arkoday Roychowdhury
El Foster
Hanna Pieniazek
Lewis Keat
Matthew Baghomian
Matthew Ford
Subham Magar

Methods & Approaches

We decided to implement an automated testing approach using JUnit testing for narrow and medium scope tests. To do so, we created a headless project within the LibGDX project and used plugins such as Jupiter and Mockito for our testing frameworks. JUnit and Jupiter were chosen specifically for their maintainability and readability, with an added benefit of these frameworks having a variety of online resources and documentation associated with them. Mockito was chosen for its compatibility with JUnit and Jupiter when implementing mocks to simulate test scenarios and conditions. Using this framework would allow us to efficiently write our tests without making our project unnecessarily large.

We also included the JaCoCo plugin to automatically generate a coverage report for our unit tests, which could be viewed via a html file. Even though we primarily focused on creating automated unit tests wherever possible, we still chose to include manual testing as part of our process for large scope testing in preparation for our user evaluation.

By using a headless project for our unit tests, we were able to thoroughly test the classes and methods within the game without interfering with the main project and source code, so team members working on the implementation and software testing deliverables could work on their respective deliverables concurrently. This was further enforced by the fact that unit tests would be written on a separate branch, named “Testing” on GitHub. Doing so also allowed us to experiment with the Gradle build configuration to make the headless project work while ensuring a playable version of the project was always available in preparation for the user evaluation. Furthermore, creating a separate branch for software testing meant that we could refactor core project classes while reducing the risk of generating build errors within the playable project. Through this approach we were able to easily detect bugs and faults within the project, improve our test coverage and reduce the amount of manual testing.

While writing and creating our unit tests, we encountered build issues within the headless project; specifically, with shader compilation when instantiating the YettiGame and GameScreen classes. We were able to mitigate some of these issues by using Mockito to create mock instances of these classes. However, in the cases where this did not work, we chose to use manual testing instead.

To ensure all requirements were covered by our automated and manual tests we used a traceability matrix to keep track of this. A Requirements ID column was also included in our Full Test Report document for additional clarity. Both documents can be viewed on the project website.

Test Report Summary

Below is a key for Test Case IDs. Naming was based on the superclass being tested and what test class the test was run under.

Test Case ID	Class Tests
Ev[abbreviated test class name]00x	Event tests
Et[abbreviated test class name]00x	Entity tests
TT00x	Timer tests
LBT00x	Leaderboard tests
Ac[abbreviated test class name]00x	Achievement tests
[Abbreviated class name]T00x	UI tests

Automated Testing

Element	Overall Coverage
io.github.yetti_eng	43%
io.github.yetti_eng.entities	50%
io.github.yetti_eng.events	77%
io.github.yetti_eng.achievements	74%

Automated Test Cases Planned	Automated Test Cases Executed	Automated Test Cases Passed	Automated Test Cases Failed
51	49	47 (95.82%)	2 (4.08%)

Test classes were structured by writing multiple tests for each specific class, with each test focusing on one behaviour or outcome. The key super classes (and their respective subclasses) tested using this approach included the Event, Entity, Leaderboard (in io.github.yetti_eng) and Achievement classes. By organising our test classes in this way we were able to tailor test cases to the expected behaviours of each class, and minimise the effect of outside factors. Doing so decreased the likelihood of writing unreliable or flaky tests.

For Event subclasses, the general approach was to test if they activated correctly, if their associated item would appear in the player's inventory, if the correct attribute for EventCounter was incremented and whether any associated achievement was achieved.

For Entity subclasses, the general approach was to test if their attributes or states would change correctly depending on their interaction with other entities and whether they would spawn under the correct conditions (if applicable).

For the Leaderboard class, the unit tests used normal and boundary test data to check whether entries would be added correctly (containing only the five highest scores). A unit test using duplicate test data was also included to ensure this would be handled correctly.

For the Achievement classes, the unit tests focused on whether their attributes and states would change correctly based on the current game states (e.g., AllEventsAcheivement.acheived is only true when all events have been activated).

Two automated tests were initially meant to be executed for the Dean entity. The first was meant to test if the Dean entity would be enabled once the player interacted with the EvidentEvent item. The second would be to test if the Dean entity would be enabled once there was one minute left on the timer. However, due to build and dependency issues, we opted to use manual testing for the Dean to save time and avoid making more changes to headless project build.

Failed Tests

IncreaseTimerEventTests (TT006) failed due to issues fetching values from the Timer class using thenReturn() methods for YettiGame and GameScreen classes. Due to time and project constraints, this was tested manually as part of a GameScreen UI test. The original unit test has been commented out of the TimerTests class and does not appear in the Jacoco coverage reports. Despite the reduced test coverage, using this approach for this class ensured that we were still able to check the core functionality of the class and that it met the requirements of the project.

Leaderboard test LBT002 failed when adding a sixth entry to the leaderboard as it did not match the model leaderboard created within the test. This was due to the Leaderboard class accessing a previous version of the file leaderboard.txt which already contained placeholder entries that did not match those used in the test class. As a result, the Leaderboard class was then refactored to generate an empty HashMap if no filepath to leaderboard.txt is given.

Manual Testing

Manual Test Cases Planned	**Manual Test Cases Executed**	**Manual Test Cases Passed**	**Manual Test Cases Failed**
9	13	13 (100%)	0 (0%)

The primary focus of our medium scope tests was on the project's UI, which included multiple screens, menus and icons (which are contained within the o.github.yetti_eng.screens package in the project). A full scope test was also included to meet the UR_RUNTIME requirement. Over the course of the project, the number of manual tests increased as any failed unit tests would be tested manually. These manual tests were used to ensure that certain aspects of the game would behave and interact with each other as intended.