Testing 2

Cohort 1, Group 6

Group Members:

Hussain Alhabib

Ellen Matthews

Minnie Poon

Jason Ruan

Daniel Smith

Owen Smith

Our Chosen Testing Methods and Approaches

Taking inspiration from Part 1, Chapter 8 of Ian Sommerville's Software Engineering book, we designed a comprehensive testing strategy that consisted of three stages; Development Testing, Release Testing and User Testing - the latter of which can be found in the User Evaluation deliverable. Out of the former two, we focused more on the development testing stage as we decided it was more prudent to ensure the game was as free from bugs and defects as possible.

To apply the development and release testing strategies to our project, we split our testing approach into two sections; Automated Testing and Manual Testing. A more detailed explanation of both can be found below.

1. Automated Testing

To make our testing as efficient as possible, we aimed to test the majority of our systems functionalities automatically. To do this we utilised a popular testing framework, called JUnit that allowed us to design automated test cases for our classes and run them during development within our project's repository. When designing our test cases, we ensured that they were both comprehensive but simple and that they had as much coverage as possible over the class' functionalities.

2. Manual Testing

For the classes, methods and interactions that we could not design automated test cases for, we conducted manual tests to ensure that they still performed as expected. These tests were also particularly useful for verifying that gameplay logic worked as expected and that the user was informed accurately during the gameplay. We structured our manual test cases in a simple, yet effective way ensuring full traceability to our requirements (related to Release testing) which can be found within the Change Report [See the Requirements Testing Index at the end of this document]. The structure was as follows; ID, Objective, Linked Requirements, Steps, Expected Outcome, and Actual Outcome. The ID of each test is related to the category it is in; UI/UX, Gameplay, Event, Difficulty, Achievement, Leaderboard, Non-Functional Requirements.

Reasoning and Appropriateness of our Testing Methodology

We chose the strategy laid out above for two key reasons; one - although more abstract, our strategy is a typical approach to testing for commercial software systems as denoted in the book, and two - we felt that it covered our game extensively.

However, to note, as our project is much smaller than most commercial software systems, we have not implemented everything laid out in the book. For example, as there are not many components within the system, we focus more on unit tests and the coverage of them over developing component and system tests as we feel these are more better represented via manual test cases.

Another reason to favour development testing is the fact that it favours tests that find bugs in the system during the debugging process and it is carried out by the team/programmer that developed the system - the same as our project.

An Overview of our Testing Results

Test Type	**Tests Run**	**Failed Tests**	**Success Rate**
Unit Tests	32	1	97%
Integration Tests	3	0	100%
Manual Tests	27	0.5	98%

Note: The JUnit Report contains 35 tests (32 Unit Tests, and 3 Integration Tests). The Integration Tests are all three of the EventTests.

A Breakdown of our Automated Tests

From our JUnit Report

Test Class	**Tests Run**	**Failed Tests**	**Success Rate**
AchievementManagerTests	9	0	100%
EventTests	3	0	100%
GameLogicTests	11	1	90%
LeaderboardTests	4	0	100%
ScoreTests	8	0	100%

Tests That Failed

As you can see above in our JUnit Test Report, out of the 35 Tests we created, only 1 failed. This was related to the timer functionality method: testGameTimerBounds(). This test ensures that there are bounds on the game's timer i.e. it remains and cannot go lower than 0 and that it can go no higher than Year 3 Semester 2. Unfortunately, we discovered that there is no code that prevents the timer from going below 0, however in context this is an extremely minor issue as once the timer reaches 0 the game redirects the player to the GameOver screen anyway and thus the player is not affected by this failed test.

From our JaCoCo Report

Test Element	**Instructions Coverage**	**Branches Coverage**
io.github.unisim.ui	0%	0%
io.github.unisim.world	7%	0%
io.github.unisim.screen	0%	0%

io.github.unisim	43%	43%
io.github.unisim.building	16%	1%
io.github.unisim.event	55%	50%
io.github.unisim.achievement	93%	85%

The Completeness of Our Automated Tests

Unfortunately, as you can see above in the summary of our JaCoCo report, we failed to achieve 100% coverage in any of our game elements. This was a combination of many factors including a miscalculation of how much we had left to do programming wise, and rather than not meet the full specification of the Assessment 2 product brief we sacrificed part of the testing. However, it is worth noting that the areas with the highest coverage i.e. those we wrote the most tests for, and those that involve the most GameLogic - this is something we did on purpose by prioritising the classes and components of the game that are the most important and had to be tested by automatic testing. For those with lower coverage, i.e. the UI and the Screen, we added extra Manual Test cases to compensate for. It is also worth noting that of the tests we implemented, only 1 failed thus highlighting the well structured foundations of our game's code.

A Breakdown of our Manual Tests

An Overview

Test Category	**Tests Run**	**Failed Tests**	**Success Rate**
UI/UX Tests	5	0	100%
Gameplay Tests	4	0	100%
Event Tests	5	0	100%
Difficulty Tests	3	0	100%
Achievement Tests	3	0	100%
Leaderboard Tests	5	0	100%
Non-Functional Requirement Tests	2	0.5	75%

Tests That Failed

As you can see above in our Manual Test Cases overview, only 1 test had a (partial) fail. The test in question was TC_PERF_01 that measured the game's performance across a wide range of devices/platforms, particularly on low-spec devices. The only reason we classed this test as having partially failed was because we could not test some of the performance aspects. We tested the game across multiple operating systems and devices but we could not accurately ascertain specific performance metrics such as the frame rate which would meet the fit criteria of the linked requirements - NFR_PERFORMANCE and CR_LOW_SPEC.

The Completeness of Our Manual Tests

Other than the partial-fail test mentioned above, I would describe our Manual Test cases as being very comprehensive and thus covering the vast majority of the front-end aspects of the game - there is no menu or screen available to the player that has not been tested.

While there are aspects of certain test cases that overlap with some of the automated tests, we felt it was prudent to include both as there wasn't a 100% overlap between them i.e. we tested manually that each of the Achievement tests would be triggered despite their being automated test cases for such, but we also checked that the user was informed that they had unlocked these achievements - something you could not test automatically.

Links to Our Testing Material

For Automated Tests:

- See our JUnit Test Results:
  https://uoy-team-six.github.io/a2/assets/testing/junit_report/index.html
- See our JaCoCo Coverage Report:
  https://uoy-team-six.github.io/a2/assets/testing/jacoco_report/index.html

For Manual Tests:

- See our Manual Test Case Descriptions (inc. Results):
  https://uoy-team-six.github.io/a2/assets/docs/Manual-Test-Cases.pdf

For Requirement Traceability:

- See our Requirements Testing Index:
  https://uoy-team-six.github.io/a2/assets/docs/Requirements-Testing-Index.pdf