Software Testing Report

Group 4

Isabella Miles
Maciek Racis
Sally Finnon
Sophia Eaves
Wrijurekh Mukherjee
Xavi Murtagh Molina

In order to ensure that the features in our game work as intended, and that the game fulfils the requirements we devised, we tested it thoroughly. To begin with, we created a plan for testing, with the tests we planned to carry out being drawn mostly from the project requirements and then being broken down into what would be testable units of code. These initial tests we devised were black box tests.

We then added some white box tests to this plan as the project developed and we realised that more features would be added, or some code needed to be tested more thoroughly. Moreover, there were some further white box tests that we added as we thought that certain blocks of code used particularly complicated logic and would need specific testing.

We aimed to use a large number of unit tests, a smaller number of integration tests, and an even smaller number of end-to-end tests, as per the Google test pyramid [1]. This would mean that the majority of tests could be automated and therefore run as many times as needed to continually test features as they continued to be implemented, saving time and effort. To avoid needing to run graphics-related code in these tests, we wrote them in a headless backend.

However, as we began to write tests, we realised that there was a significant amount of logic that required creating an instance of a class that runs graphics-related code, meaning we could not write tests for this code in the headless backend. We also attempted to use the Mockito framework to mock the implementation of the game, but we were not able to use this to write tests requiring UI-related classes either. We therefore had to carry out these tests manually, as end-to-end tests, and thus could not achieve the ideal proportions to unit, integration and end-to-end tests given by the Google test pyramid [1]. Nevertheless, we were still able to write a large number of unit tests.

In order to write unit tests, we used a framework called JUnit, of which we used version 5. JUnit allowed us to use a variety of convenient methods, such as AssertTrue and AssertEquals, and tags such as @Test and @BeforeEach. Additionally, we used JaCoCo for examining test coverage. JUnit and JaCoCo are both well-known frameworks for testing in Java, which our game is written in, so we decided they would be appropriate to use.

Moreover, we devised a number of manual end-to-end tests to cover both what we had originally intended to test manually, and also what we discovered we could not automate whilst writing unit tests. Although it was not ideal to have the proportion of manual tests that we did, it was a feasible amount for a small scale game like this. Furthermore, as the game only lasts five minutes and is not excessively difficult, manual testing can be carried out on it with not too much trouble.

During the process of designing our tests in more detail, we carefully considered what input values we would use for both automated and manual tests. We thought about what input values would give what expected values, dividing the input domain into equivalence classes. We were then able to select values from each equivalence class, making sure to include not only values well into the equivalence classes, but also values on the boundary of different classes. Moreover, we made certain to test invalid inputs as well as valid ones.

With regards to automated unit tests, we wrote a total of 56 tests, with every test passing. These consisted of code testing achievements, the existence of every asset, building counters, buildings, game over data, money, with Point class, the Score class, student satisfaction, settings, and the timer.

The test summary of passing unit tests can be viewed on our website at this link: [insert link].

These tests cover 26.2% (17/65) of classes, 16.7% (54/324) of methods, 10.7% (51/477) of branches and 11% (194/1756) of lines in total. However, the majority of the code not covered is code that could not be tested in the headless backend due to requiring graphics, or is code relating to the structure and setting up of the project.

The classes that automated tests could be written for, or had code run by automated tests for other classes, are ReputationClass, MoneyClass, GameOverData, Achievements, Point, AchievementManager, Timer, Leaderboard, MusicManager, Test, Settings, Score, HeadlessLauncher, BuildingManager, BuildingType and Building. This excludes Main, GameState, FullscreenInputProcessor, WorldInputProcessor, World, UiInputProcessor, StartMenuScreen, SettingsScreen, Popups, LeaderboardScreen, InfoBar, HowToPlayScreen, GameScreen, BuildingMenu, AchievementScreen, ShapeActor, GameOverScreen, StartupHelper and Lwjgl3Launcher.

Of the former classes that we were able to write automated tests for in the headless backend, 94.4% (17/18) of classes, 55.7% (54/97) of methods, 40.5% (51/126) of branches and 56.6% (194/343) of lines were covered. This gives a better picture of the code coverage of our unit tests based on what was possible to write tests for.

However, although it may have been possible to write some automated tests for the Leaderboard (excluding code that is run when testing if the Leaderboard.txt asset exists) and MusicManager classes, we opted to test these manually for the sake of convenience.

In MusicManager, the code that could be tested was mostly trivial, such as getters and setters, and other code would not be possible to test in the headless backend. Furthermore, it would not be possible to test whether the music would actually play and at what volume as an automated unit test - this would be an end-to-end test - which is the main requirement of the MusicManager class.

In addition to this, although automated tests could technically be written for the Leaderboard class, these would require manually clearing the leaderboard.txt file every time they are run, defeating the purpose of automated testing. We therefore opted to carry these tests out manually.

Excluding these two classes, as well as Test and HeadlessLauncher, which are classes concerned with writing unit tests rather than actual game logic, we had a class coverage of 100% (14/14), a method coverage of 65.8% (48/73), a branch coverage of 52.1% (49/94) and a line coverage of 67.6% (173/256). The remaining code that was not covered consists mainly of basic getters and setters, and methods in the BuildingManager class and AchievementManager class that could not be tested or could only be partially tested due to requiring graphics.

The test coverage report can be viewed on our website at this link: [insert link].

We originally had a lower test coverage after, but evaluating what parts of the code were being tested and what parts weren't enabled us to discover more parts we could be testing. We therefore modified our test plans and wrote tests for these parts of the project, to better ensure the completeness of our code.

We performed tests that could not be automated manually, by running the game, carrying out a sequence of actions to achieve the desired input, and observing the output. These tests were taken from our original plan based on our requirements, with some tests being ones we had planned to be manual from the start and others being tests we found we were not able to automate. To keep track of manual testing, we created a table recording each test, with a reference, description, the steps to be followed, the expected outcome, the actual outcome, the status of the test and what requirements it relates to.

In total, we performed 119 manual tests, covering, for example, the functionality of events, the timer, buildings, money, general performance, navigating between various screens, the leaderboard, achievements, difficulty, the pause button, the button to play again, the camera, the sound and music, the map, scoring, that various screens display correctly, that the player can enter their name, that the game ends correctly, that you can mute the game, that you can zoom in and out, and that the game runs on Windows, MacOS and Linux.

Although these tests overlap with some of the classes tested by unit tests, they functionality that could not be tested in the game's headless backend. To give an example, we used automated tests to test that achievements are unlocked when the player has completed the task required for each achievement, but had to manually test that an achievement displays on the game screen when it is unlocked.

Out of the 112 manual tests we performed, 107 of the tests passed and 5 of the tests failed, meaning that 95.5% of the tests were passed. The tests that did not pass are as follows:

18) Test that building counter displayed on screen correctly
58) The camera cannot move past certain boundaries
86) The volume is displayed correctly
87) The difficulty is displayed correctly
107) Initial changes to volume are saved when the game starts

TEST 18: This error occurred during the process of adding more buildings. Although the 4 buildings used for part one of the project (eating, sleeping, food and recreation) fit nicely and clearly on the information bar at the top of the screen. When adding more buildings, e.g.," recreation 2" their counters were pushed down, spilling off the bar and into the game screen. This was messy and hard to read. To fix it, alternate versions of buildings were all labelled under the same type. To reduce the number of counters and conform with the requirements. For example, the normal and previous student accommodation are both counted as "sleeping" Consequently everything now fits. The error was identified on Tuesday 07/01/2024 and fixed by Wednesday 08/01/2024

TEST 58: This error enabled the players to move the camera infinitely, in any direction.

Causing the main map to get lost. The error was identified on Saturday 11/01/2024 and fixed later on the same day by implementing a boundary feature to restrict camera movement.

TESTS 86 AND 87: Was an error that occurred when updating the settings screen look from a temp, to a final version. The labels for difficulty and the volume slider got mixed up. The error was quickly fixed by swapping the variables around. So volume and difficulty labels are now correct. The error was identified on Saturday 11/01/2024 and fixed later on the same day

TEST 107: Although muting the volume works initially, the error occurs when moving from the main menu to the game screen by clicking play. Despite the slider retaining its position and saying the volume has been muted, music can be heard. This error was identified on Sunday 12/01/2024 and fixed later that same day by reworking the music manager class to accommodate for the changes in state. Ensuring that the volume status is checked before starting the in-game music

[1]T. Manshreck, *Software Engineering at Google: Lessons Learned from Programming over Time.* Oreilly & Associates Inc, 2020.