Testing

Cohort 1 Group 10

Cai Hughes<cabh500@york.ac.uk>
Ben Slater<bs1463@york.ac.uk>
Adeola Adeniji<aa3098@york.ac.uk>
Mathew Riedy<mr1723@york.ac.uk>
Riad Kasmi<rmk526@york.ac.uk>
Simon Konieczny<sk2144@york.ac.uk>

Introduction

To test our game we used a combination of unit tests, manual tests, and full run throughs of the game. Each approach was chosen based on its appropriateness for testing the class or function, allowing for a flexible testing environment that would compliment our system design.

To begin our testing we split the types of functions into different groups, mainly focusing on game logic tests and graphical tests. This allowed us to appropriately design tests based on the functionality that was needed.

Functions that were based on game logic were unit tested. This included our activity calculations, player movement, day, time, and energy calculations, and leaderboard functionality. Each unit test tries to isolate the tested components to help figure out what might be causing problems in development if they occur. To create isolation we used Mockito to create mock instances that did not interfere with the main game system. Our unit tests were automated to run as part of the build process, ensuring that there was continuous validation of the code at every change made. This allowed us to catch bugs early before they became a problem later and ensure that all new code did not break existing functionality.

Manual testing involved us interacting with the software directly, playing either the whole game or segments of the game that were pertinent to what we were testing. This was mainly used to test our graphical functions in the game, like the map display, animations, menu screens, start and end screens. Some functions, while not directly tied to screen animations, were unable to be completed in unit tests as they needed graphics from libGDX to load, and these had to be tested manually as this was not possible in our testing environment. Manual testing was also used to give insight into the user experience. While a unit test can tell us whether or not a function is performing properly, it can not tell us whether it is intuitive or fun to play and this was also a component of our game. We used varying approaches to testing, from predetermined actions in order to see functionality that we were changing, as well as exploratory testing to identify unexpected behaviours such as glitches or errors that could hamper the game experience.

Unit Tests

Automated Testing File

Our unit tests had small coverage of all our functions, with only 15%, however with specific classes we had very high coverage as these were mostly focused on game logic which were able to be unit tested.

StatsTracker

Our statstracker class is mainly focused on how the score of the player is calculated, containing functions to increase scores and calculate grades and streaks.
Our tests were designed to test the effect of adding sleep, relaxation, eating points. Retrieving those same points from a statstracker entity. Setting the max streaks from these scores and checking what the highest achieved streak is. And initialising a StatsTracker to ensure that it can be properly instantiated. The functions that we did not test were render, setup and dispose functions which draw scores to the screen.

Leaderboard

Our leaderboard class needed to store a player's name and score. If there were more than 10 scores stored it needs to only store the top 10. To devise tests for this class we needed to assess whether we could add to the leaderboard, save the leaderboard, load the leaderboard and instantiate a leaderboard. We were able to pass all of our tests. Showing a full functionality of our testing.

CameraManagerTests

While not all of our camera manager could be unit tested, we were able to test the initial positions, the boundaries of the camera, the logic of updating the camera position and zooming in and out. These tests allowed us to look at the logic of our camera, and see if there were any unexpected behaviours. All our tests were passed, but further testing would be done manually.

DayCycleManagerTest

We were able to unit test a limited section of our day cycle manager. This allowed us to test whether we could cycle through the seven days that we needed to complete. Whether we could add time when an activity was completed, and whether we could access the day and time. All of these are key to the games functionality and we were able to fully pass our tests

MenuTests

We tested whether a render call could be made, so that we could confirm that the functionality would allow us to display the menu screen. The rest of the class relates to displaying graphics so was tested manually.

PlayerTests

For our player class it is important that it can move in any direction, and whether it will stop when it has collided with a surface. We created unit tests to ensure that this class was fully functional and we were able to completely cover this in every direction that the player is able to move.

EnergyMeterTest

Our energy meter’s logic needs to be able to reset the energy bar at the start of the day, add to and remove from the energy bar. We were able to test this functionality and pass the tests. The rest of this class's functionality was manually tested.

Manual Testing

Manual Testing File

Manual Testing was used to test the functionality of visual systems which were either inefficient to automate testing for or were not able to be manually tested, they were conducted by our group with input about the functionality and the general feel and so were more qualitative than our automated testing.

CameraManager

The movement of the camera following the player and the ability to zoom in and out was tested by players to ensure that it functioned correctly and felt enjoyable to play. The main method for this was to run between buildings in the game, while changing the values to find a comfortable position that did not have any glitches.

DayCycleManager

The day cycle manager draws the day and time to the screen, this was checked by quickly sleeping through the days to ensure that there were not any typos. As well as letting the game timer progress through a full day to ensure that it was functioning correctly.

EndScreen

The end screen consists of tables drawn to the screen. This functionality was tested by getting to the end screen to ensure that everything was correctly drawn. There are also ClickListener functions that were independently tested to ensure proper functionality.

EnergyMeter

The energy meter as it is drawn on the screen was tested manually, by going to activities, we could calculate what the outcome of an activity would be on the energy meter and then confirm whether it was behaving correctly.

Game

The game contains the music that is played, we listened to the music file before loading the game to confirm that it is not being distorted and is playing correctly. It also draws the menu screen, while not explicitly tested, it would indicate a failure if there was no menu screen and the game would not be able to progress.

GameMap

The game map could not be tested by our unit tests, so we had to do manual testing. This involved various tasks throughout the map that interacted with the class's functionality.

GameScreen

The game screen draws a lot of the game's visual elements to the screen. These had to be tested manually as they could not be unit tested. To do this we listed the functions and created tests for each of them to ensure they were working correctly.

InGameMenu

The in-game menu is drawn to the screen, and produces results that are easily visible so it made sense for this to be tested manually. It was able to pass these tests.

MenuScreen/SettingScreen

Similar to both the in-game menu and menu screen, these were easy to assess visually, with clear effects when buttons were pressed and clear layouts on the screen. This made manual testing a clear choice and useful for debugging potential problems.