Software Testing Report

Cohort 1, Group 5
Team Name:
JAzZ MoLeS
Group Members:
Sophia Taylor, Lucy Wood, Mitchell Gilbert
Jamie Creed, Archie Adams, Zayed Iqbal

Testing Methods And Approaches:

The testing methods we will be using were all inspired by the week 4 testing lecture delivered on test doubles, and for the documentation of our tests were inspired from a post within the same website listed during the lecture[1]. When creating tests we will be focusing on test cases which involve checking dependencies between functions if they existed, erroneous data which'll throw an exception valid data which the functions were designed to use.

To avoid scenarios where bugs were caused we added test cases which handle this for example a scenario where the players energy is negative which shouldn't be possible. We will also have to continue testing the new features that get added. To make sure our test coverage is continuously improved we decided to add Github Actions, Junit Tests and JaCoCo to our workflow to help test coverage be constantly improved and be added to our CI workflow. To make sure there's traceability between the requirements and testing we have decided to use Google Sheets to record our test cases taking this inspiration from the blog post.

Unit tests are mainly being used to ensure that anything new added does not break our current code functionality so these are being applied to test each function of code and these are being applied to test each function within classes thoroughly. We did dive into integration and functional tests but considering the scale of our project as its all one Java project it would not be appropriate since different applications aren't being used and also its not suitable considering the limited timeframe we have.

For manual testing our approach was to list step by step instructions on how to recreate the test done, what we hope the outcome would be and saying if it's been achieved with a pass or fail. By recreating the steps we can accurately find out if a feature is working or not. This is a useful guideline to features couldn't be unit tested

Brief testing report:

The majority of our tests were done with unit testing, trying to achieve a 100% we tried automating tests where it was plausible. However these could not be done for graphics.

Graphics within games often have a lot of complex interactions and behaviours that are quite difficult to capture with unit testing alone. Manually testing allows us to experience the game

from the player's perspective making sure that interactions feel intuitive and responsive, So we have chosen manual tests for these so any missing coverage should be ignored and they are:

- UI responsiveness:
    - StartScreenInput -KeyDown(int), nextscreen(),StartScreenInput(HustleGame)
    - EndScreenInput -KeyDown(int), EndScreenInput(HustleGame)
    - ScoreScreenInput -KeyDown(int),ScoreScreenDown(HustleGame),finish()
    - ScoreScreen -ScoreScreen(HustleGame)

- Visual feedback:
    - GameScreenUi -showHud(), showPlayer(), showMap(), update()
    - ScoreScreenUi -initScoreLabels(), ScoreScreenUi(),update()
    - EndScreenUi -positionBottomLabels(Map, int, int, int), EndScreenUi(HustleGame), positionTopLabels(), update()
    - StartScreenUi -StartScreenUi(), update()

The manual tests cover key functionality like game loading, energy and time management, day transitions, scoring system and leaderboard functionality. While the manual tests confirm the primary features work, additionally they should be aided with additional tests to cover edge cases and possible failures.

Delving into the automated unit tests the coverage for instructions was 39% and for branches it was 61% over half of them covered. All the tests have successfully passed which indicates that under the tested scenarios the current implementation is stable. As said before some coverage involves graphics like GameScreen, StartScreen, EndScreen. Most of them have been covered by manually testing them. Even with tests like 'gamestate' and 'scoring' even while they are well tested they can still be further improved upon. It is unfortunate that there are still some areas of the code coverage that aren't covered by either the automated unit testing and manual testing which means that there might still be some hidden potential bugs.

Not all branches are covered by unit tests like:
if(tempNodeMap.containsKey(playerPosition)){returntempNodeMap.get(playerPosition).getActivities();}
This if statement produces 2 branches in the JaCoCo test report. However, the only code it affects is a block if the statement returns true. Due to time constraints, we have created tests for the main scenario:

- When the entire statement is true (the map contains the key and activities are returned).

We have not written unique tests for every permutation of the condition's truth values. This makes the "branches" section of the JaCoCo report look slightly lower. Nevertheless, the code surrounding and within the if block is adequately covered by the unit tests.

The JaCoCo coverage report is shown below:

https://st1835.github.io/assessment2/jacoco/test/html/index.html

The test evaluation can be found below:

https://st1835.github.io/assessment2/tests/test/index.html

Google Sheets pages describing our unit tests and manual tests can be found here:
Testing Spreadsheet

Currently all the tests that have been described in our testing documentation have all passed. We have achieved as much coverage as we possibly could via JUnit and where it wasn't possible to use JUnit we did manual testing.

Reference:

[1] Oleg Romanyuk. How to write test documents: why you need them, and how to get started [Online]. Available at: https://medium.com/free-code-camp/how-to-write-test-documents-why-you-need-them-and-how-to-get-started-b17440823007 [Accessed: 15 May 2024]