Software Testing Report

Cohort 1, Team 3

Ari Kikezos
Arya Enkhnasan
Ben Green
Calum Wright
Lilac Graham
Skylar Garrett

Testing Methods

Testing Method 1: Automated Testing

The easiest kind of testing to prove that the program was in the correct and functioning state was automated testing using JUnit. Along with Coverage Testing (described more in CI2), automated tests allow the development team to ensure that any changes made to certain parts of the software do not break other parts of the software inadvertently. If these tests fail when they didn't previously, this is a breaking change and can be fixed in line with the whole program, not just the new feature. The results of this testing are available in full on the website, as linked to at the bottom of the document.

Testing Method 2: Manual Testing

As well as any automated tests that could have been and were performed, it was also necessary to run the code on our own machines to make sure that design or implementation errors which were not programmatic issues (errors that didn't crash the program or were as simple as logical errors) were also caught. Broadly, this encompassed things such as layering and rendering issues. This was, in part, the reasoning behind performing user evaluation (described more in Eval2); errors that cannot be caught by machines sometimes cannot be caught by the developers either because they have too intimate an understanding of the function of the program.

Testing Results

Out of the tests performed on the whole codebase, by the end of development, tests had a success rate of 100%. This accounted for 12% of the codebase. These results are available in more detail on the team's website. These tests are so few because of development problems with getting headless to work on the codebase for reasons none of the team could fathom. These are the details of the tests performed:

Testing Results: Main Program

The main program was difficult to test due to aforementioned limitations, but every static element (static variables, static functions, etc.) runs without issue. This section of the testing suite covered 30% of the available code.

Testing Results: Systems

Systems testing was more successful.

Testing Results: Systems: Toast System

This section of the suite tests the following and covered 100% of the system:

- That toast of both the default colour and custom colour can be added.
- That existing toasts can have their colour changed.
- That toasts that have expired get removed from the list.

There were no problems detected with this system.

Testing Results: Systems: Leaderboard System

This section of the suite tests the following and covered 68% of the system:

- That entries could be added to the system.
- That entries were sorted correctly.
- That the lowest score could be popped manually.
- That the input would be un-JSONified.
- That the output would be JSONified.

Problem areas were:

- The leaderboard system did not recognise the file path in the testing environment. This was accounted for by playing the game to completion and then again and finding that the leaderboard does, in fact, behave as expected in the final version.

Testing Results: Systems: Other Systems

Some tests exist to test other systems, but no detail was applied to them due to aforementioned headless complications.

Testings Results: Completeness of Testing

Of course, the automated testing is not complete, far from it. To compensate for this, we put lots of effort into manual testing towards the end of the project, and the user evaluation was invaluable to make sure that any obvious game breaking errors would be fixed before the final product was released.

Bibliography

Jacoco Coverage Testing Results:
https://nsq511.github.io/ENG1-TIV-Part-2/assessment2/jacoco/test/html/index.html
JUnit Unit Testing Results:
https://nsq511.github.io/ENG1-TIV-Part-2/assessment2/tests/test/index.html