UNIVERSITY OF YORK
DEPARTMENT OF COMPUTER SCIENCE

Testing
Engineering 1 - Assessment 2

Group 2, Cohort 1 ("*The JVMs*")

Ben Hatch
Charlotte Sharp
Dan Nicholson
Ethan Andres
Freddie Higham
Joe Silva
Rosie Kern

JVMS

Testing methods and approaches

Upon our inheritance of the project, our team developed automatic unit tests for the existing implementation of the game. This involved using the JUnit library to write test methods which check whether the inherited code behaves as it should. The tests pass different valid and invalid input data into the methods being tested to ascertain whether the expected output data are produced. Testing the inherited code was important because we as a team needed to make sure that the additional code we would go on to develop would not be placed on top of unstable or incorrect foundations.

Following the writing of the unit tests for the existing implementation, our approach to developing the game further focussed on writing the tests for our new code prior to actually writing the code. This paradigm of test-driven development would force us to consider the requirements of our code before implementing the solution, and would limit unnecessary code as we would be focussing on fulfilling the test criteria. [1]

Writing the test methods with JUnit integrated neatly with our IDE (IntelliJ) and our build automation tool (Gradle). This allowed unit tests to be run either all at once or individually by pressing a button in the IDE. Having this speed of testing enabled the team to pass tests quickly and to move on with further development within our limited development time. We were also easily able to run Jacoco coverage tests using Gradle which gave us an indication of where more tests would be needed.

As some aspects of our game could not be easily tested using automated tests, some areas were manually tested by a member of the team. This involved writing a set of criteria that an aspect of the game needed to pass to fulfil our requirements and then manually playing the game to observe whether the criteria were met. An example of one of these manual tests was to check that the player's avatar would move across the map in the correct directions when the control keys were pressed. Testing this automatically with unit tests would have been much more time-consuming than simply observing the behaviour with a manual end-to-end test. To follow the Google testing pyramid, we aimed to limit the number of end-to-end manual tests so as to keep the majority of our tests small and automated by way of unit tests. [2]

Automatic Testing Report

Our Jacoco coverage report indicates that 47% of our code was covered by automatic testing. While this is a relatively low percentage, we believe that we have tested as much of the logic via unit tests as possible. Our "Classes Covered List" on the website shows which of the classes we tested and how much of their functionality was tested. We also have a "Functional Requirements Tested" table which traces our test classes to the corresponding functional requirements which they test.

None of the "Screens" classes or "render systems" were tested due to them being mainly graphics based. A lot of the Screens classes consisted of instantiating entities of classes where we had already written automated tests for, so this did not need to be tested again. Similarly, the input system was only partially tested due to us not being able to simulate player keyboard inputs. The "Constants" classes were deemed unnecessary for testing due to their simplicity (only containing values for constants). The Collision system wasn't tested due to it being too complex to test via code and because a lot of its logic was handled by a trusted library (Box2D).

As a result of our project using the Ashley Entity-Component system, the "component" and "system" classes were hard to test and, in some cases, weren't fully tested. For the components, we wrote tests which checked whether each component could be successfully added to an entity. Given that, in Ashley, components contain almost no logic of their own, it made no sense to test them any more than their ability to be added to entities. The "system" classes had more logic to test, though oftentimes that logic was relating to rendering, collisions and player input which we decided were better to test manually. We decided to test if the systems could successfully be added to an engine and if they were able to identify the appropriate entities (for example, could the "InteractionSystem" find all of the interactable entities). These tests allowed us to be certain that the entity-component architecture had been set up correctly but didn't necessarily show that the full logic of the system was functioning.

What we did fully test was the important logic such as score calculation, streaks and the leaderboard. For the score calculation, we tested many different scenarios, one where the player completes an inadequate amount of activities, one where the player spends 2 consecutive days not studying, one where the player doesn't eat enough etc. These scenarios covered almost all of the possible score boundaries the player could achieve and allowed us to verify that our calculator would always award the player with an appropriate score. Similarly, for the leaderboard we tested the writing and reading of files as well as the logic for adding player names and ranking them accordingly. In general, we considered the intended functionality of each of these classes and tried to write tests which would prove the methods functioned as intended no matter what parameters were inputted. This included searching for edge cases and deciding on different bands of test cases.

In our current implementation, 100% of the tests have been passed successfully. This is because we have been continuously iterating upon our code such that it passes the tests (in-line with the test-driven paradigm). We have also been modifying the test code where problems and bugs have been identified so that the tests are as accurate as possible. For example, we had an issue once where some of our score tests were not being passed and we identified an error with the test code which we then fixed.

Manual Testing Report

The automatic unit tests were extremely useful for testing many aspects of the implementation, but there were some areas where unit tests could not fully test the functionality. The main areas we weren't able to automatically test are collisions, animations, movement, interactions and menu buttons. Therefore to ensure we achieved full coverage with our testing some manual tests needed to be carried out.

To perform these manual tests a table of instructions with an 'expected output' was created, to then be followed by the tester. The main areas of testing are as mentioned previously and with 19 tests carried out those areas are now adequately tested.

With collision testing there were no issues found relating to the actual physics of the collisions, so at no point could the player walk through the buildings or walk off screen. However there was one area where the collision did not match the visual boundary of the building, so in that aspect the test did fail. As a team we found this failure to be very minor so no change has been carried out so far, however to fix this we would need to change the vertices of the collision to match the building better, which would be simple.

The animation testing failed four tests, these failures were all to do with the diagonal movement not having the expected animation. Initially it was expected that the player's animation would match the horizontal movement, however we found it instead matches the vertical movement. After consideration, we agreed this failure does not affect the gameplay or cause any confusion and therefore is not seen as a serious problem. To fix this would involve changing the name of the sprites used when diagonal movement is performed.

Aside from the animations related to the movement, the movement testing itself had no failures. The testing involved all 4 main directions, up down left right, along with each of the diagonal movements and in each case the player entity updated its location accordingly to the direction and speed provided. These 8 tests clearly cover every form of movement present in the game.

Interactions also needed to have manual testing, mainly the relationship between pressing 'E' and having the appropriate counters update. This test included every single interaction present on the map and checked every counter each time and no failures were detected. As every location and every counter was checked, these tests completely cover all areas of the interactions.

Finally the menu buttons and screens were manually tested. This involved pressing the 'start', 'main menu', 'leaderboard' and 'quit' buttons, and observing the screen changes afterwards. Each test in this area passed successfully, however there were assumptions made that the same button functions the same even on different screens. This could be incorrect, therefore the coverage of this area is not quite complete.

Overall almost every test in the manual testing section passed, with every failure being a minor issue that can be dealt with easily in the future. Every area that needed to be tested has been given enough tests to be covered with a high level of completeness.

References

[1] Bluefruit. (2019, Feb. 22). *Unit Testing and development: Build tests before you code*. bluefruit.co.uk. [Online]. Available at: https://bluefruit.co.uk/people/unit-testing-build-tests-before/ [Accessed: 6 May 2024].
[2] Waker. (2015, April. 22). Just Say No to More End-to-End Tests. testing.googleblog.com [Online]. Available at: https://testing.googleblog.com/2015/04/just-say-no-to-more-end-to-end-tests.html [Accessed: 9 May 2024].