TESTING REPORT

Dalai Java

Team Members

Jack Kershaw, Max Lloyd, James Hau, Peter Clark,
William Marr, Yuqing Gong

Testing Methods and Approaches

In this section we will cover the methods and approaches the team has taken to tackle testing our game thoroughly. In general, as mentioned in the other reports, we split our team into speciality sub-groups. These sub-groups were to focus on specific parts of the game and the project. Thus, our testing strategy applies to those in the sub-group, with frequent discussion with the development team. However, this all starts with a strong test strategy, which is essential for a well-structured testing process.

For our strategy the whole team worked together to decide which roles go to whom; what platforms and frameworks are to be used; the risks and mitigation; and the schedule that testing will run on. After evaluating the relative strengths of the team, we assigned roles as follows: Peter took charge of writing Unit tests, Max was in charge of evaluating the syntax and logic of existing unit tests and that these tests provided maximum code coverage, and Yuqing was in charge of performing manual tests. The schedule was complex due to the ideal test practice - which is to write tests before implementation. This allowed the development team to test their code to ensure it follows the overall game requirements. However, we did not want the development team to have to wait for platforms and frameworks to be integrated and then tests written. We decided that the tests will be created alongside the code developed, and set a specific testing deadline which was a week before our official deadline. The platforms and frameworks that we used for the testing were JUnit and Mockito. JUnit is a very popular framework that is robust and well tested. This gave the group assurance that the passing and failing unit tests were not flukes. Then for mocking we decided to use Mockito, it is widely and professionally used, which gave the team confidence. For the risks, the largest was the time to learn testing to an effective level. Not all members of the testing team had done Unit testing, and none had done mocking. The only mitigation to this was for the testing team to take a week to focus on honing their testing skills. Another risk was new, unexpected, code that could be implemented by the developer team. This would be managed via two methods; regular meetings between the two teams, and for every pull the testing team member did - to read the changes made through IntelliJ's merge system. Overall the testing strategy worked well with the small team that we had for testing.

The overall goal of the testing team was to ensure all requirements of the game worked as intended. Thus, the functional testing approach was appropriate, since it allowed us to directly check these requirements. This is done through creating input data based on the functions purpose, entering it, and checking if the output is as intended by comparing to the expected outcome. The key is to isolate each requirement, ensure it is testing only one thing, make it repeatable, fast, timely and self-validating (this is where JUnit is especially useful). Then to ensure all possible combinations and types of data are accounted for, we used boundary and parameterised testing. This is where we see the boundary of input data for a function, for example, and test around said boundary. We test right outside the boundaries, on the limit, just inside and an average value. This is not needed for all functions, this would take too much time, but it is crucial for functions that are vital to the game. We set our Stop Criteria to be when 100% of the methods were covered either by Unit or manual tests.

As editing the code for Assessment 3 required a moderate amount of refactoring, we needed to ensure that functionality which was already present did not fail. To ensure this, we carried out regression tests; each test which had been passed in Assessment 2 was re-tested to ensure that existing functionality within the game was not affected by refactoring and updates. This can be seen in the Status (A3) column of our Test Documentation.

Brief Report on Actual Tests

Overall, we ran 73 tests - this consisted of 50 Unit tests from 8 classes, and 23 manual tests.

Unit Tests

We started our Unit tests from the top of the UML Class Diagram, with the Entity class, a simple class with no complex functions. It is, however, the class that all other in-game objects are extended from, so it was imperative to test it thoroughly and correctly. We then continued by testing the other classes in the class diagram from the top down.

A brief report of the Unit tests can be seen below:

Class	Tests Ran	Tests Passed	Tests Failed	Line Coverage
Entity	3	3	0	78%
Unit	10	10	0	92%
Projectile	5	4	1	83%
Fortress	2	2	0	46%
Character	2	2	0	85%
Firetruck	14	14	0	80%
Alien	10	4	0	71%
Pipe	6	6	0	76%

The one failed test relates to our decision to test out of boundary values, such as negative numbers, since we cannot allow Entities outside of the allotted game board space. To check this we used a more primitive method of a try/catch since most team members understood this better than Lambdas, which are supported in JUnit5. Most of these tests passed by the end of the project, however the negative number check was a failure. This is because the implementation of catching negative numbers would cause the game to crash. Due to later implementation, Projectiles would end up going to negative positions before we called *dispose()* on the object.

Our unit tests varied in terms of completeness; some of our tests boasted very high line coverage, whilst others had a considerably lower value. For some methods the figure given is unrepresentative; the Fortress class, for example, has 100% method coverage, however the majority of the lines of code are assigning a number of 'aliens' to positions based on the level. Similarly, for the Firetruck class, certain functions change very slightly due to the parameter passed to them, resulting in lower line coverage when only one instance of the function has been tested. This results in some uncertainty considering the completeness of our Unit tests, however as the majority of methods have been tested, we believe that our tests are relatively comprehensive. However, the Unit tests were unable to cover the entire code due to dependencies on classes which could not be mocked using PowerMock, hence we had to rely on manual tests for a considerable amount of our testing.

In terms of correctness, with Unit tests there is a chance that tests can be incorrect as only the output of methods is tested as opposed to the method itself, meaning that correct results may have

been output simply by chance. However, we used both boundary testing and parameterized testing to ensure that the risk of this occurring was minimal.

Manual Tests

There were cases where the testing team ran into problems, where we did not know how to test certain requirements with the previous method, including **UR_start_screen**, **UR_select_level**, **UR_pause**, and **UR_music**. To test these User Requirements, we conducted manual tests in which we ran through the game, and manually checked that these User Requirements were working. In total we carried out 23 manual tests; 13 of these related to the gameplay of the game, whilst 10 related to the 'Preferences' saved in the background of the game.

One of our manual tests failed; this was test 10.5.2, testing whether the outcome of the minigame had any effect on whether the Fire Engine was refilled or repaired. Initially we were unable to pass this test due to the nature of the GameStates; they are initialised on a stack, making it difficult to pass any results down from one state to another before popping. We then discovered a partial solution to the problem, however the test still failed as when the minigame is completed once, it is marked as completed the second time regardless of whether it is completed or not.

As our Traceability Matrix shows (see link below) our combined Unit and Manual tests covered all of our User Requirements, therefore our tests can be seen to be complete. It is slightly more difficult to prove whether our manual tests are correct due to human error, however having tested each feature multiple times this has solidified our belief that our manual tests are correct.

In terms of regression testing, each test which had been passed previously was tested once again and once again passed, giving us confidence that refactoring and improving our code did not affect functions already in existence; the only test to fail the regression test was the test which had failed previously as we were unable to find a solution to this bug.

Overall, the testing was fairly complete, with most line coverage over 60% and method coverage as well. With some classes being lower due to mostly being getter and setter type functions. The testing done by the team gave the game a more concrete code base that will hopefully satisfy the client.

Testing Design and Evidence of Testing

Test Documentation: https://dalaijava.github.io/files/test_documentation.pdf

Traceability Matrix: https://dalaijava.github.io/files/traceability_matrix.pdf

Unit Test Traceability: https://dalaijava.github.io/files/Unit%20Test%20Traceability.pdf

The tests themselves can be found in the ZIP file, under core -> test.