Module	Engineering1 (Eng1) - COM00019
Assessment Title	Assessment 2, Cohort 2
Team	Dragon Boat Z (Team 18)
Members	Robert Dalgleish, Benjamin Jenner, Joseph Lonsdale, Richard Upton, James Wilkinson, Xinyi Zhang
Deliverable	Testing Methods and Approaches

Testing Methods and Approaches

In our first meetings, following the newly updated requirements, we considered our overall testing strategy and began designing our testing plan. During planning, it was clear that we would need to consider several factors:

- What testing techniques will be used and at what stage of the development
- What purpose do these techniques serve
- Where time will be spent amongst these testing techniques

Testing our software continuously throughout development aims to ensure that the final product meets as many requirements as possible and catches as many bugs as possible that detract from the intended experience of the product. Multiple testing methods are required to test all the various aspects of the software, these include **unit**, **integration** and **system** testing.

Whilst developing our testing strategy, we decided that we should split the group into two subgroups, where one would focus on testing and the other would focus on the implementation of new features.

We designed **Black Box** tests using our requirements to identify any missing features in the game. This could then be used in development to target changes that needed to be made to the original codebase. During the design of these tests, we looked at valid, invalid and edge case inputs to ensure as many situations were covered as possible, without exhaustive testing. A traceability matrix was then used to check that all requirements were tested thoroughly, to reduce test redundancy, and to identify test cases to change as requirements change. As the initial Black Box test was based on the initial state of the inherited game, an additional Black Box session was run on the complete product to re-test previously failed cases and test the newly implemented features.

White Box testing was then designed to be used throughout development. The advantage of White Box testing is that it can be carried out automatically and quickly, allowing new changes to be quickly checked. We decided to use the JUnit framework to write our tests, as this integrates well with Java and can be used to generate test reports. Mockito was used to mock objects; this was required where certain classes were dependent on others. In order to keep these tests independent and break dependencies between other classes, mock objects were used wherever possible and documented properly.

An **agile** approach was used for testing to fit with the development environment. This allowed us to create new tests for newly developed features, and ensure that at all times, as much of the codebase as possible, has been tested.

Test coverage was measured using modules of the IntelliJ IDE. We decided to use line coverage, as statement coverage wasn't an option, and method coverage, for a broader view.

Unfortunately unit testing does not work for all aspects of the product, especially for system testing, and some modules such as the UI can't be tested effectively with White Box. This is why manual testing was still required to look at the system as a whole and for some integration tests.

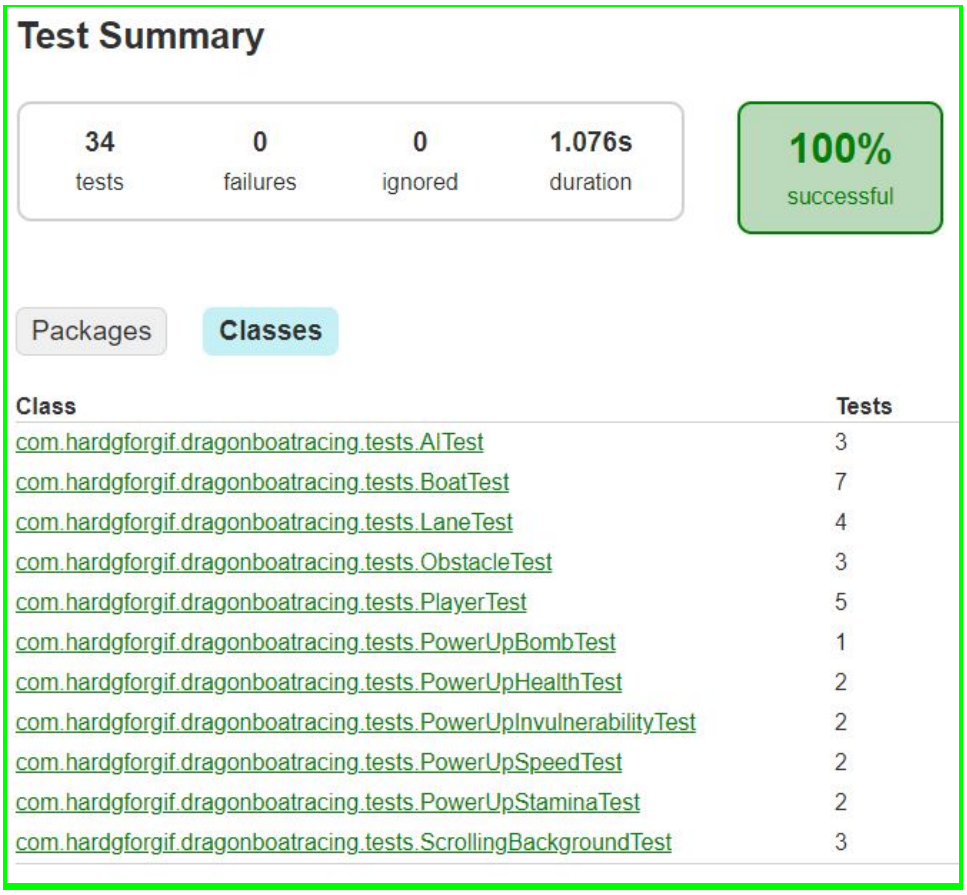
Due to this, Black Box testing was carried out again at the end of development, to measure the success of the implementation of our requirements.

If exhaustive testing is to occur, this can cause the Pesticide Paradox to come to light. This states that 'for methods used to find bugs, these can leave residue of more subtle bugs which testing for is ineffectual'. To avoid this, a termination criteria was set. This helped to avoid subtle bugs and to prioritise the tests that were most relevant.
This termination criteria was decided on a class to class basis: if the method and line coverage were above 80%, or continuing with testing was deemed infeasible due to inter-object dependency, the testing development would stop.

Testing report

White Box

Our White Box testing included all unit tests and some of our integration tests. 34 test methods were carried out on 12 classes with a pass rate of 100%. A zip file with the generated test report can be found here titled as Test Report.



Test Summary

34 tests	0 failures	0 ignored	1.076s duration

100% successful

Packages | Classes

Class	Tests
com.hardgforgif.dragonboatracing.tests.AITest	3
com.hardgforgif.dragonboatracing.tests.BoatTest	7
com.hardgforgif.dragonboatracing.tests.LaneTest	4
com.hardgforgif.dragonboatracing.tests.ObstacleTest	3
com.hardgforgif.dragonboatracing.tests.PlayerTest	5
com.hardgforgif.dragonboatracing.tests.PowerUpBombTest	1
com.hardgforgif.dragonboatracing.tests.PowerUpHealthTest	2
com.hardgforgif.dragonboatracing.tests.PowerUpInvulnerabilityTest	2
com.hardgforgif.dragonboatracing.tests.PowerUpSpeedTest	2
com.hardgforgif.dragonboatracing.tests.PowerUpStaminaTest	2
com.hardgforgif.dragonboatracing.tests.ScrollingBackgroundTest	3

Coverage of Individual Test Classes

The average line coverage (not including the Map and PowerUpBomb classes) is 83.3% line and 82.5% method coverage. The reason we haven't included the "Map" class in unit testing is because the headless LibGDX environment used to carry out these tests ran into errors when trying to run this code, making it impractical to White Box test. The coverage for "PowerUpBomb" is low because the implementation of the class was incomplete, making it difficult to test beyond the constructor. This means that for the rest of the modules, the target of 80% coverage was met and the White Box testing can be deemed to be adequate for all but the "PowerUpBomb" and "Map" classes.

Class	Method Coverage (%)	Line Coverage(%)
AI	84	66
Boat	67	70
Lane	71	64
Map	0	0
Obstacle	76	76
Player	66	74

PowerUp	66	75
PowerUpBomb	50	11
PowerUpHealth	100	100
PowerUpInvulnerability	100	100
PowerUpSpeed	100	100
PowerUpStamina	100	100
ScrollingBackground	78	91

We believe that these **Unit Tests** cover as much of the core functionality as is feasibly possible, without exhaustively testing each class. The reason for lower line coverage for “AI” was due to all but two of the methods being private, this made achieving high branch coverage near exhaustive, due to the “updateAI” method. Instead we decided it would be best to cover these tests with Black Box. “Lane” also has low line coverage due to high dependency on other classes that couldn’t be mocked easily, again we decided this would be best to test with Black Box.

For **integration testing**, the “PowerUp” classes used the “Boat” class in their unit tests, allowing us to test these classes working together. Another aspect of integration testing was the saving feature, but this was carried out by manual tests instead due to there being too many dependencies and the “Map” class being required. Therefore we decided carrying out these tests manually would be more suitable.

Black Box

To guarantee that System Testing provided tests for all requirements and that they had been met, a traceability matrix was constructed to match System Tests to the Requirements. This helped to ensure the completeness of these System testings This was formatted so that it could be clearly seen the number of tests that were relevant to a requirement and vice versa. This allowed for easy identification of cases where Requirements had not been tested. The Traceability Matrix can be found here.

We conducted an initial Black Box testing, with our new requirements, after inheriting the existing project. This was to provide insight into missing features; new requirements already met and additional bugs that had not been identified. The link to this can be found here titled as Black Box Testing - Initial. This returned a pass rate of 48% (75/155) and helped us to identify what should be prioritised during further implementation and debugging of the code base.

During our second round of Black Box testing, found here titled as Black Box Testing - Final, we achieved a 92.9% pass rate (144/155).
These results revealed issues with our current implementation and what would need to be changed to pass all these tests and meet all our requirements. The failed tests can be seen here FailedTests.

- 1.8.1 - Tests for a help button in the main menu that takes you to a screen that shows basic information about the controls of the game. This test couldn’t pass, as the feature wasn’t implemented in the final build.

- 1.12.1 - Tests if the AI achieved better times with increased game difficulty. It failed as the AI have better stats and can move faster, but the increase in obstacles negated this factor and led to AI boats having worse times on average, although fastest AI times did increase with difficulty.
- 1.14.1 - Tests the Powerup Bomb, which removes all currently visible obstacles in the boat's lane. This test failed as the feature is not yet fully implemented and so the powerup is in the game but its effect does nothing.
- 2.5.1 and 2.5.2 - Test for a mini-map showing the course and player's location and that the map updates respectively. These both couldn't pass, due to the Minimap feature not being implemented in the final build.
- 2.8.2 - Tested that exiting from the pause menu caused a pop-up prompting the user to save first. This failed due to the feature not being implemented in the final build.
- 2.13.4 - Tests that the music loops on the pause menu. This failed due to the audio track failing to loop after finishing in the pause menu.
- 3.2.7 - Tests that the transition from pause menu to the leaderboard is less than 0.5 seconds. This fails as the screen goes blank when this action is carried out.
- 3.4.2 and 3.4.3 - Test that the Geese and Log obstacles, respectively, are animated smoothly. This failed due to the sprites not being animated, as the feature was not implemented in the final build.
- 3.5.9 - Tests that the Minimap is recognisable to the user. This test couldn't pass as the Minimap feature was not implemented in the final build.

In order for these tests to be passed, the missing features in the final build would need to be implemented. Additionally, AI stats need adjusting (1.8.1); the PowerUpBomb class would need to be completed (1.14.1); music looping on the pause menu needs to be added (2.13.4) and resuming to the leaderboard needs fixing (3.2.7).