Module	Engineering1 (Eng1) - COM00019
Assessment Title	Assessment 2, Cohort 2
Team	Dragonite (Team 21)
Members	Omar Omar, Rhianna Edwards, Okan Deniz, Craig Smith, Omar Galvao Da Silva, Joel Wallis
Deliverable	Testing

Software Testing Report

In order to fit the testing plan with the overall development plan, Testing was divided into three phases. The goal was to make a simple plan where testing is a crucial part of the development cycle. By dividing the testing process into three phases, an agile testing environment was established. This allowed for easy allocation of resources and allowed for greater flexibility. The order in which the testing took place was also important.

Phase One: Static and Dynamic Analysis:

Testing and inspection through the direct execution of test data in a controlled environment and human evaluation of code, design documents and modifications. Weekly code reviews were conducted to ensure that the current development cycle meets coding standards of certain quality. Verification and Validation were carried out at the end of each review.

Justification: Based on research, it was determined that the first phase would focus on initial code review with low levels of detailed automated testing. While weekly reviews were ideal, as they provided developers with quick feedback on their work, verification and validation were essential to make sure that any necessary changes to the requirements and design were made. As such, a blend of both strategies was implemented. Incorrect design or implementation decisions could then be identified promptly.

Phase Two: Black Box Testing:

Also known as system testing, Black Box testing does not require knowledge of underlying implementation to derive the test cases provided in the requirements specification. Equivalence class testing and Boundary Values testing were used to determine the input space while User case testing was used to simulate user system interaction. Requirements were read and input/output variables were identified. Keeping in mind how input variables can influence the output, equivalent class analysis was performed to explore the boundaries of these classes.

Justification: Black Box testing was done before White Box testing to avoid spending too much time on optimising code that did not meet the requirements. Equivalence class and Boundary Value Testing were used to ensure that the input space was adequately tested without having to test every single input. A mix of manual and automated tests were used to limit human error. Lastly, user case testing was used to mimic the typical interaction between the user and the system. This kind of testing allowed for exploration of fringe cases where a user might face a problem.

Phase Three: White Box Testing:

Structure of the source code was used to guide testing through creating tests that satisfy obligations and measuring the coverage of existing tests. Statement and Branch coverage were used to ensure the code is efficient.

Justification: The third phase mainly focused on code optimization through code coverage. Aiming for a high code coverage meant that any code that was not used was to be removed. If significant parts of the program were not tested it meant that the testing was inadequate. This helped verify that adequate test inputs were used for existing tests. White box test cases can also be easily automated and it’s efficient in finding errors and problems for better optimisation.

Testing Summary:

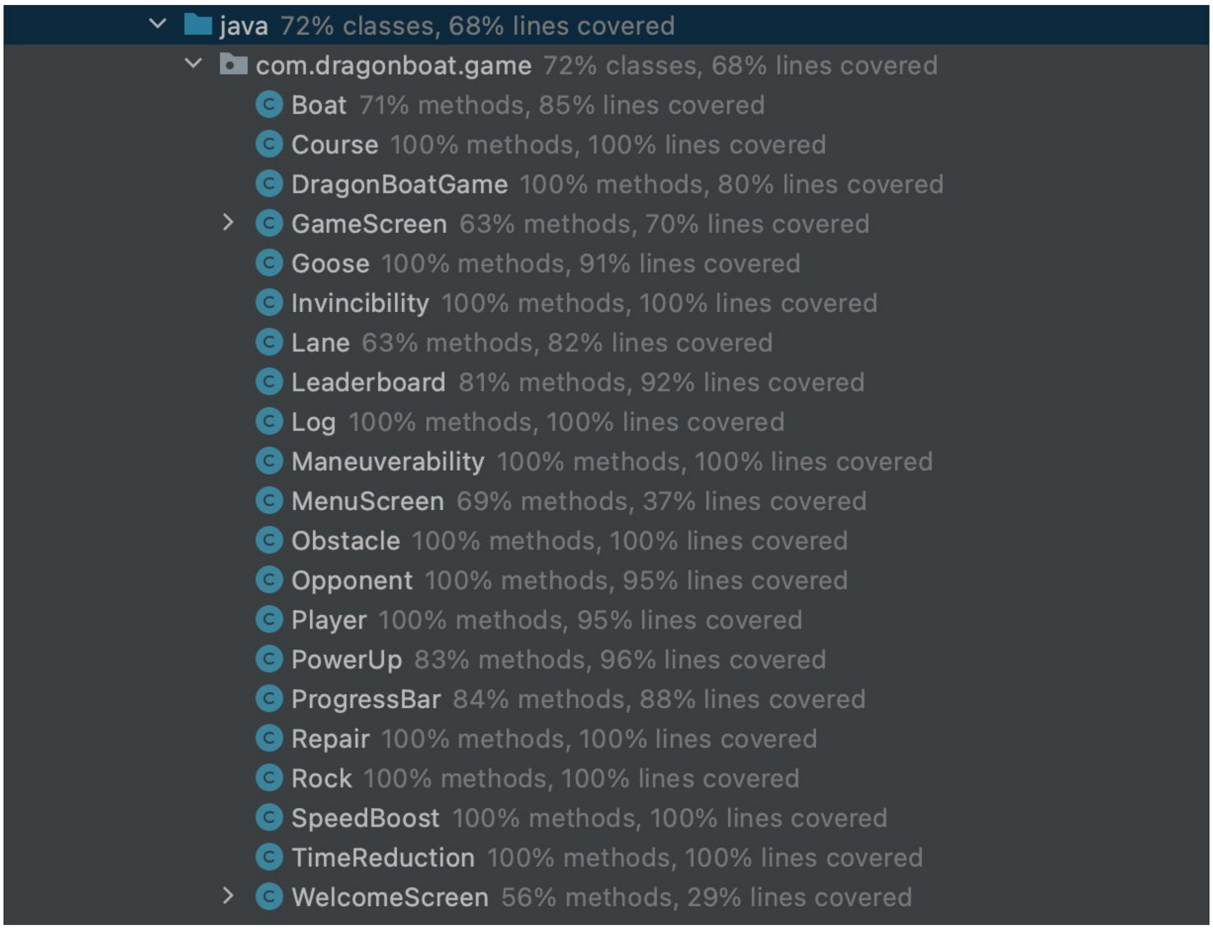
When we started testing, we divided the tests according to their corresponding requirements. This can be in the traceability matrix linked below. The matrix made it easier to track our progress and divide the work among us. In the matrix, requirements ,where the test case was the same, were grouped together. This led to a cut down in the number of tests that needed to be done. Every test case is explained in the test table linked below.

The tests run by the team were adequate to test the requirements. However due to time constraints, the team was unable to perform all the white box testing needed and reach a higher code coverage. As such, the team focused on testing the requirements and making sure that the main classes work. This decision was made to avoid wasting time on running tests on Subclasses that were almost identical to each other such as Log and Rock. We also recognise that most of the code coverage obtained came from running the manual tests which vary based on the circumstances. However, it should also be noted that most of the game code involves using a sprite batch to print textures on the screen. Such code cannot be tested without a manual test. The statistics on the tests below highlight this.

The table provides a description, input, owners,comments ,expected and actual values of each test. The material for these tests is linked below. All the automatic tests can be found under “Automatic Tests”. However, manual tests have individual links to their videos. To make it easier to identify which link corresponds to each manual test, the link title is the same as the test’s requirement

Testing Statistics:

- 100% of the tests passed
- Code Coverage after running all the tests:



```
java 72% classes, 68% lines covered
  com.dragonboat.game 72% classes, 68% lines covered
    Boat 71% methods, 85% lines covered
    Course 100% methods, 100% lines covered
    DragonBoatGame 100% methods, 80% lines covered
    GameScreen 63% methods, 70% lines covered
    Goose 100% methods, 91% lines covered
    Invincibility 100% methods, 100% lines covered
    Lane 63% methods, 82% lines covered
    Leaderboard 81% methods, 92% lines covered
    Log 100% methods, 100% lines covered
    Maneuverability 100% methods, 100% lines covered
    MenuScreen 69% methods, 37% lines covered
    Obstacle 100% methods, 100% lines covered
    Opponent 100% methods, 95% lines covered
    Player 100% methods, 95% lines covered
    PowerUp 83% methods, 96% lines covered
    ProgressBar 84% methods, 88% lines covered
    Repair 100% methods, 100% lines covered
    Rock 100% methods, 100% lines covered
    SpeedBoost 100% methods, 100% lines covered
    TimeReduction 100% methods, 100% lines covered
    WelcomeScreen 56% methods, 29% lines covered
```

Testing Material:

- Traceability Matrix: https://omar-h-omar.github.io/ENG1-Dragonite-Assessment-2.github.io/docs/deliverables2/Traceability%20Matrix.pdf
- Automatic Tests: https://omar-h-omar.github.io/ENG1-Dragonite-Assessment-2.github.io/tests/build/reports/tests/test/index.html
- FR_ANIMATIONS: https://drive.google.com/file/d/1Z40W5EGmP1z-CrYO3SMFUAfms1kF1jEk/view?usp=sharing
- NFR_RESPONSIVE, FR_LANE_WARNING: https://drive.google.com/file/d/18zbvhOotlZxy3vwk1QbMPCt6oR6D8M4o/view?usp=sharing
- NFR_GAME_LENGTH: https://drive.google.com/file/d/1vYBJqsw-IelFq-VL3l8dbfYN1NvA0scF/view?usp=sharing
- UR_LOSS Video 1: https://drive.google.com/file/d/1wKLCaI6HuEvTMhfNugCtZ5RYo7gdMgXL/view?usp=sharing
- UR_LOSS Video 2: https://drive.google.com/file/d/1fE9bUOzZpsyA8Jf6V2dL58-3sYPpEoIR/view?usp=sharing
- FR_PAUSE_SCREEN: https://drive.google.com/file/d/11ZfilukqzsrelLmkH83TYtXHfQ4lM-q1/view?usp=sharing
- FR_TITLE_SCREEN: https://drive.google.com/file/d/129sGoDf6_R4ToL4roovvD_yA1qlDihZv/view?usp=sharing
- FR_GAME_MODES Video 1: https://drive.google.com/file/d/10hUcJ6FXW32HjWjEWTf2lw56puNZ8kt-/view?usp=sharing
- FR_GAME_MODES Video 2: https://drive.google.com/file/d/16SRjU4R_25ihwKkW58MLmehslHQBpbcW/view?usp=sharing
- FR_GAME_MODES Video 2: https://drive.google.com/file/d/1tvJMG3b8T3-Uxtl0Xx4Th8wrJja9mrMQ/view?usp=sharing

. **Resources Used:**

- https://junit.org/junit5/docs/current/user-guide/#overview-getting-started
- http://techduke.io/junit-testing-of-libgdx-game-in-android-studio/
- https://stackify.com/best-software-testing-methods/
- https://www.perforce.com/blog/alm/8-key-software-testing-methods
- https://www.perforce.com/blog/alm/black-box-testing-dead
- https://www.westagilelabs.com/blog/why-is-software-testing-and-qa-important-for-any-business/
- https://www.guru99.com/white-box-testing.html