Software Testing Report

Team CASEUS

(Team 7)
Eliot Sheehan
Matthew Turner
Daniel Atkinson
Hannah Pope
Mayan Lamont
Divyansh Pandey

Testing Methods:

Testing Approach
For our project we will only be testing public methods as private methods can only be used by the class they belong to so they can be considered implementation details which, due to our black box testing approach, can be ignored. Furthermore, we will only be testing code that we deem to have complex Logic, methods such as getters, setters and constructors will be ignored. To complete the testing we will be using a libGDX's headless backend combined with Mockito and JUnit.

Black Box Testing
Due to the relative simplicity of our project, we are less concerned with the inner workings of our methods, therefore the majority of our tests will be black box tests rather than white box tests as we only care that the output is correct for a given input.

Valid, Boundary and Invalid Testing
When appropriate, we will be testing multiple different inputs to the same methods. More specifically, we will be testing:

- Values that fall approximately in the middle of the accepted values - with the expected output being successful
- Values that fall at the upper and lower boundary of the accepted values - with the expected output being successful
- Values that fall slightly beyond the upper and lower boundary of the accepted values - with the expected output being unsuccessful
- Values that fall far outside the boundaries - with the expectation being that the method will handle it appropriately, whether that be throwing a related exception or circumventing the error that would be caused

Approach Justification
The reason that we take this approach is due to the nature and size of our program. Few types of software give the user as many opportunities to reach boundaries as games, this is why we have decided to make sure that we are always testing boundaries when suitable. Players will always look for ways to break the game and that's why we need to take into account all the eventualities we can see, because the objective often changes from winning to breaking the game. As for why we have elected to focus on black box testing rather than white box, it is quite simple: while there are many methods throughout the game code, most are simple. There is little point testing every internal aspect of a method when that method is five lines long. Furthermore, due to the fact that java is statically-typed, we can usually say with assurance that the method will be able to handle what is thrown at it.

Actual Tests:

Boat Tests

The majority of tests done were done on the boat class. This is because the boat is what the player controls, therefore this is also where most of the opportunities to break something are. Overall we have 25 tests for the boat class that can be broken down into seven groups:

- Movement
  - These five tests simply checked that if the player or AI tried to move a boat in a certain direction, that it would move in that direction, provided they were not at the boundaries of the course
- Speed
  - Two of the four tests checked that the speed increased or decreased when told to, the other two were boundary tests that make sure the speed doesn't surpass the max speed and the speed can't fall below 0
- Damage
  - One test checked that the method worked and the other checked that a possible error case is handled
- Boosts
  - The three tests check that Boosts are applied and removed correctly and that no errors will be thrown if you try and remove a Boost that isn't there
- Lane checks
  - These five tests covered all possibilities for the lanes: one test checking that the boat being in the acceptable region is fine, two boundary tests checking that the boat being at either limit is fine and two boundary tests checking that the boats being just beyond either limit is not fine
- Fastest time updates
  - These five tests check that the fastest time updates correctly and takes the penalties the boat accumulated into account
- Reset
  - One simple test to make sure that the reset method changes the values to what we expect

Obstacle Tests

The Obstacle class was the next class to be tested, but this one had relatively fewer tests as theoretically we can predict every event that may occur as the player has no control over the Obstacles. Furthermore, the Obstacle class is used more as a framework for the different types of Obstacle rather than functioning on its own. Overall there were two tests. Firstly, we tested that when told to move the Obstacle does move from its original position. Then we also made sure that if the Obstacle was removed that it was removed successfully.

Goose Tests

The next class to be tested was Goose, which is an extension of the Obstacle class. Again, there aren't many tests as we should be able to mostly predict the behaviour of objects of this class. First we test that the Goose can successfully change the direction of its movement when told to, and then we checked that the Goose actually moves when told to.

Lane Tests

The next class we tested was the lane class, this is an important class in the game as each instance of lane tracks and manages the Obstacles spawning in that lane. Therefore we have five tests for this class. The first test we have checks that when a new Obstacle is spawned in the lane the Obstacle count for the lane is increased successfully. Then we went on to check that when spawning an Obstacle of a certain type (Boost, Goose and Log all extend the Obstacle class) the right type of Obstacle is created, i.e. when we try and create a Log a Log is created rather than a Goose or a Boost. Finally, we tested that when the Obstacle limit has been reached, any attempt to make an Obstacle is halted without throwing an error

ProgressBar Tests

We then went on to test the ProgressBar class. This class tracks how far through the race all the competitors are and whether the race is finished or not. The first test we did was a reset test, we needed to do this as for each leg the progress bar will need to be reset and this needs to be working well. We then did a test that checks whether the progress bar increments successfully as this is another core aspect of the progress bar. Then we went on to run four tests to make sure that the race deemed finished prematurely, we made sure that the player finishing had no more significance than any other boat. Then we checked that the method that returns the progress of the boats as an array of floats between 0.0 and 1.0 returns the correct values and never surpasses 1.0 even if some boats have surpassed the finish line.

IO Tests

Used in saving, this class controls the interaction between the game and the file where the game is saved. The first thing we tested was that we are correctly and successfully writing and reading to and from files. Here we also checked that if a file didn't exist reading it simply returns null rather than throwing an error. Then we checked that we were successfully converting between JSON and usable data in both directions.

JSON Tests

For a large chunk of the classes in this project we have two methods essential to saving. Namely, we have a toJSON method and a make*classname* method. These methods are used to convert the instance of the class into JSON when saving and then back into game data when loading. The classes that have these methods are: Player, Opponent, Goose, Log, Boost, Lane and ProgressBar. Therefore, we have two additional tests for each of these classes that check the JSON is being translated correctly. Furthermore, the DragonBoatGame class also has a makeDragonBoatGame method that recreates the whole game upon load, but unlike the other classes mentioned above it doesn't have a toJSON test.

Test Comments:

All of the tests I mentioned above pass successfully bar two. This is likely due to the fact that as we went along writing the tests, any tests that didn't pass and highlighted errors in the code, simply resulted in us fixing the code. We were testing and fixing as we went along. This does bring into question how complete our testing was, as apart from very small programs there are almost certain to be bugs in any piece of software. The only conclusion that I would draw from this is that we are limited due to the fact that we were involved in the creation of the software. If we wanted to do further testing we would likely get an independent party to check across our code without editing it or run some alpha and beta tests to try and find bugs in the gameplay rather than the code. Furthermore, sometimes the bugs aren’t easy to detect as the code technically works, just not in the way we intended it to. In terms of our correctness I think our tests are simple enough that we can say with some degree of certainty that the tests are working fine, again any error here is more likely to be human error where we are testing in a manner different from what we intended.

The two tests that did fail, did so because the data they needed was never saved. We know how to fix this issue, but due to the fact that the project has reached the end of it’s time window we do not have the resources to do so.

Testing Table Link:
https://caseus7.github.io/Dragon-Boat-Z/docs/deliverables2/TestsTable.pdf

Test Results Link:
https://caseus7.github.io/Dragon-Boat-Z/docs/tests/index.html