Assessment: 2

Deliverable: Testing

Team Name: Team 8

Team members: Charlie Hayes, Matilda Garcia, Joshua Stafford, David Kayode, Ionut Manasia, Matthew Tomlinson

Introduction

When testing this project we tried to automate as many tests as possible, however when testing the requirements of the project we found that it would take up valuable time and resources to automate requirements testing, this is because for the size of the game it would take more time to research and set up automated testing than it would to just manually test them. For that reason we decided to split our testing into white box testing (which would be automated) and black box testing (also called requirements testing, which will be completed manually)[1].

For white box testing, the method of choice was the use of automated unit testing, using JUnit 4 as the framework. We also needed the GDXTesting framework, made by github users TomGril and BeeGodwin, that allowed us to create a mock libgdx testing environment. The tests have been focusing mostly on the classes in the core package of our game, as those were the ones that had complex logic involved in the functionality of the game. For each tested class, a test class has been created with a corresponding name inside a testing package. Methods that contain no logic might have been used while creating the test variables, but methods that are only used in the rendering of the said class have been skipped entirely. Of the methods tested, where there could have been multiple results, separate tests have been created for each of the result, except for when the said results was a boolean value, in which case a single test method was used encapsulating both results, to make sure that we avoid running into a false positive or false negative.

Due to an abundance of lines and methods in the code dedicated to the UI Elements and the creation and rendering of said visual elements, a brief estimate of the maximum coverage we could obtain was closer to 50%. An attempt was made to get the highest possible coverage in classes that are made mostly of logic that could not be easily tested in black box testing. We have also tried to create most of the tests as unit tests, so that the results would be conclusive to each of the elements being tested, but due to how some methods have been programmed, whenever it was not possible, an integration test was created instead. For example, while testing the AI, most methods would call separate methods in their class, and we would also need to initialize variables from outside the class and incorporate them into the tests. Rather than creating refactoring the code, that would be time consuming and could lead to unforeseen problems, we have decided to test them as is in integration tests.But were variables could be isolated and still come to a certain result, unit tests have been programmed instead, up to the limitation of using a mock ApplicationAdapter as our test environment.

On the other hand, for black box testing we made a test plan that contained every test to do. Each test consisted of a test ID; description, the reason for completing the test; related requirements or components, the requirement that the test is based on; criticality, how important that test is, also affects the order that we will complete the tests in; category, whether the test is a unit, integrated or system test and also whether the test is a normal or edge case; author, the person completing the test; input data, the data that will be imputed to complete the test; expected outcome, the outcome of the test that the tester expects; actual outcome, the outcome that occurs when tested; status, whether the test passes or fails and evidence of the test happening.

Test Report

For white box testing, we covered 46% of the lines in the code, and 40% of the methods, but the numbers jump up to 81% of the lines in the core package, containing the classes for non-UI in-game elements with 79% of their methods tested this is shown in the coverage report on the team website[4]. This was done in 21 tests, of which only one was constantly failing and another one is showing inclusive results. For black box testing all 27 user, functional and non-functional requirements where tested in 39 tests with 2 tests failing

The coverage report for white box testing has been saved in a html file, showing the percentage of lines, methods and class covered from each package alongside the exact numeric value. The file can be accessed from our website.

Test: The test that has failed had the name **testPlayerInputBoth** and it was meant to test if the boat would not change its direction of movement if both directional inputs binded to the keys A and D were pressed at the same time.
Reason for failing: the first condition in the parent method **updatePlayer**, (if (pressedKeys[1])) only account for the “A” key being pressed before it moves the player 90 degrees to the left, this way, when signals from both keys are received, it’s that first condition that takes place, and then the statement stops.
Fix: A simple fix was applied, by making the first two branches of the statement check if the opposite directional key is not being pressed.

Test: The test which results vary with around a 2 to 1 success rate is **testAwareness**. The test has been designed to test if an AI opponent is able to spot an obstacle that has been rendered close to him. In the preparation for this test, an AI boat has been created alongside an obstacle, and both were moved close to each other, with the obstacle x position being in front of the boat. It seems that the AI has trouble recognizing said obstacle as a threat in some cases.
Reason for failing: The reason for failing is still unknown. Due to the nature of the method being tested, it could be a range of possibilities, one of them could stem from the initialization of one of the variables that are part of the test. This shows the shortcomings of integration tests, when we cannot isolate the variable from their testing environment. Another possible reason could be the parent function itself. It contains a conditional statement that references another function, that was programmed to create a target for the AI telling it where to move. It is possible that the said function might not create the right target for the AI controlled boat to be aware of the obstacle. This is also unlikely, as the function has been tested in subsequent tests and it seems to be working optimally.
Possible fix: A possible fix is still unknown, but some that have been tried were implementing a loop that would continuously call for updates, and it passing the test whenever the condition is met at least once. This has shown to be ineffective , as the results would stay the same all across the loop leading more credibility to it being a possible quirk of the testing environment.

When designing the black box tests we had to decide some criticality to assign each test, if the test is derived from a user requirement, the criticality is set to the same. If the test is derived from a functional or non-functional requirement the criticality is set to the

corresponding criticality of the user requirement the functional or non-functional requirement is based on. If it is based on two user requirements then the one with the highest criticality is chosen. Criticalities can be reduced by the discretion of the tester if it was deemed too high or low for the given test. Tests with higher criticalities were done before tests with low ones to make sure that the essential parts of the game worked as intended allowing us to fix them rather than waste time on testing minor aspects of the game.

Of the tests that failed in black box testing none of them where failed because they didn't work completely, they only failed as they did not meet the requirements they were based on. The main one that failed was test T15 which was used to test the requirement UR_SAVE 1.3.5 below is an excerpt from testing showing test T15 (for space purposes the evidence has been removed, but it can be found in the testing section of the team website[3]):

ID: T15
Description: test whether you can save the game half way through the round and reload the save upon reloading the game
Related Requirements and Components: UR_SAVE 1.3.5
Criticality: High
Categories: System Test (Normal Case)
Author: Charlie
Input Data: load the game, play a leg and then play half way through the next leg and save. Then reload the game and see if it starts in the same position
Expected Outcome: upon reloading the player will be in the second leg and the opponents, obstacles and player boats will all be in the same position.
Actual Outcome: the player can save their round number but not the locations of their boat, the obstacles and the other boats.
Status: Fail
Possible Fixes: Extend the saving implementation to cover the serialisation of boat and obstacle objects so their state/attributes can be stored and loaded into the physics world.

As seen above saving was not completed to the standard set out by the requirements which were as such: Previous leg results, player health, stamina, position and boat should all be saved. Currently only previous leg results and current leg are saved. Full saving was not completed due to time constraints and also due to the fact that it would require refactoring a lot of the code that was handed to us at the beginning of the project. Therefore a more rudimentary version was implemented. Allowing us to focus more on the other new requirements such as difficulty and power ups.

The other failed test was T3.B. Below is the test plan for test T3.B with some information removed (as before, the rest of the information can be found in the testing section of the team website[3]):

ID: T3.B
Description: test the edge cases of the controls
Input Data: try and turn the boat 360 degrees to the left, then to the right
Expected Outcome: the boat should only be able to turn 90 degrees each way

Actual Outcome: the boat can only turn 90 degrees in each direction from its starting rotation, once the boat reaches 90 degrees it keeps moving forward as if it were pointing forward
Status: Fail

As seen above this test was an edge case so it failing does not impact the project a great deal and is the only failed test that could actually be counted as a bug. As this bug was found very late in the development cycle there was not a lot that could be done about it as the research that would have to be done into why it occurs would take too long. Upon quick analysis the problem could be down to a conditional statement in the method Boat.moveBoat() starting with: **if** (boatSprite.getRotation() < 90), this statement apparently decides the direction vector of the boat based on the player boat's head and is the most likely cause because of it.

Some requirements have multiple tests covering them, they are split into, for example, test T1.A and test T1.B. Both will be testing for the same requirement but they will each be testing different cases, be them normal or edge cases, to ensure that every possible eventuality was tested we had to make sure to test all these cases. An example of one of these requirements with many related tests is UR_POWERUPS 1.3.4. There are 4 test cases for UR_POWERUPS 1.3.4, 2 normal cases and 2 edge cases, and these tests show off the detail put into making sure that all possibilities were covered in testing.

Traceability Matrix

[https://eng1-team-8.github.io/Dragon-Race-Website-2/Assessment2/testing/Traceability_matrix.pdf]

As seen on the traceability matrix we have covered all requirements with at least one test, we used the traceability matrix to make sure that no requirements were overlooked during black box testing ensuring that we got as much test coverage as possible. It also allowed us to have no test redundancy when designing out tests making manual testing as efficient as possible.

Testing Material

All: https://eng1-team-8.github.io/Dragon-Race-Website-2/Assessment2/testing.html

Testing evidence and design:
https://eng1-team-8.github.io/Dragon-Race-Website-2/Assessment2/testing/Testing.pdf

Traceability Matrix:
https://eng1-team-8.github.io/Dragon-Race-Website-2/Assessment2/testing/Traceability_matrix.pdf

Coverage report:
https://eng1-team-8.github.io/Dragon-Race-Website-2/Assessment2/testing/coverage/index.html

References:

[1] M. Aniche, “Software Testing: from Theory to Practice” Section 2.1 [https://sttp.site/chapters/testing-techniques/specification-based-testing.html]

[2] A. J. Offutt, “Software testing: from theory to practice,” *Proceedings of COMPASS ’97: 12th Annual Conference on Computer Assurance*. doi: 10.1109/cmpass.1997.613216.

[3] C, Hayes, “Testing” [https://eng1-team-8.github.io/Dragon-Race-Website-2/Assessment2/testing/Testing.pdf]

[4] I, Manasia, “Coverage Report” [https://eng1-team-8.github.io/Dragon-Race-Website-2/Assessment2/testing/coverage/index.html]