Software Testing Report

Team 15

Joe Wrieden

Benji Garment

Marcin Mleczko

Kingsley Edore

Abir Rizwanullah

Sal Ahmed

Software Testing Report v0.3

Preface

- v0.1 Summarised our testing method and approach, explaining why these are appropriate for the project
- v0.2 Brief report on the actual tests, including statistics of what tests were run and what results were achieved, with a clear statement of any tests that are failed by the current implementation.
- v0.3 Document revised and completed.

Testing Methods and Approach

Software Testing is used to verify and validate a program. Our goal was to maximise the number of bugs found given our resources. Due to limited resources and since exhaustive testing is not feasible, we needed to prioritise the tests we needed to perform. We broke down the testing process into planning tests, and executing test cases, as per standard Software Testing procedures. We planned our tests by designing test cases, which consisted of systematically devising different test cases extracted from our Requirements specification (found here), along with asserting correctness according to various test inputs (both valid and invalid). Software Testing took place throughout the duration of the Software Engineering lifecycle, and continued for over a week after we had finished implementing the additional requirements. White Box testing is what we focused on while we were implementing the requirements, and we began Black Box testing soon after.

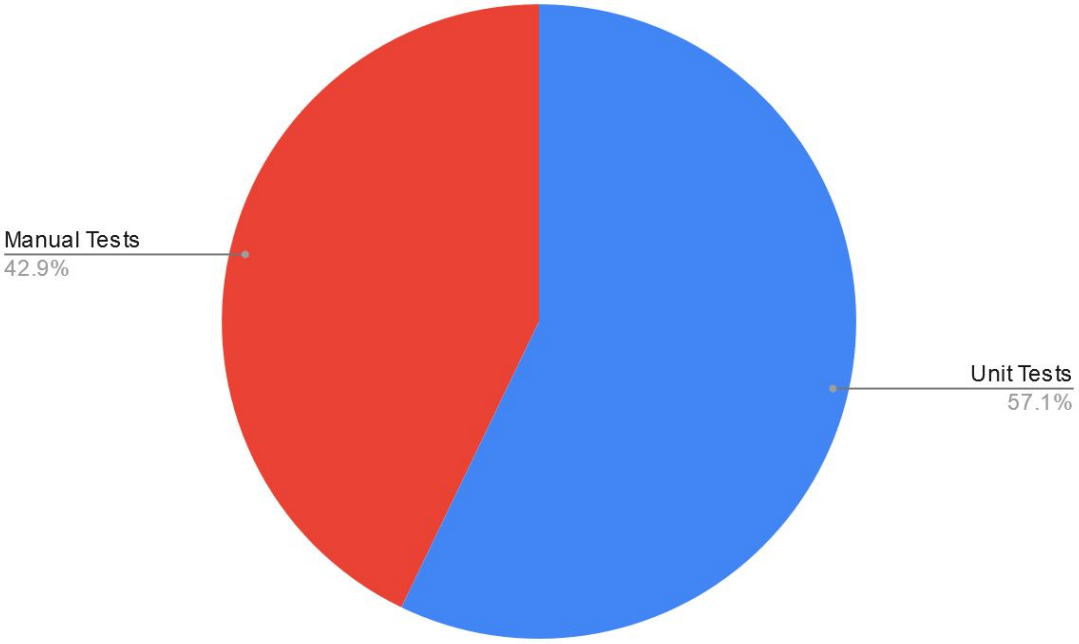
Unit testing allowed us to test the correctness of huge parts of our code by splitting it into smaller ‘units’ (with different classes/methods having a respective unit test) in isolation, while also being fast and easy to write. The Unit test methods assert that the actual output matches the expected output. Changing unit tests’ input and expected values are also straightforward [1]. Our execution of test cases was automated, as detailed in our Continuous Integration Report, which can be found here.

To begin our dynamic approach to testing by using unit tests, we used the JUnit testing framework as it is the most popular and therefore the most robust framework for Unit testing with Java. Additionally, this testing framework enables us to write test cases in a way that they can be seamlessly executed by the machine. Furthermore, as LibGDX was used to develop the game, we used a testing skeleton for LibGDX projects written by Tom Grill [3]. This skeleton implements JUnit tests and Mockito (used for mocking real components when the system is not complete) in conjunction with the LibGDX Headless Backend Application in order to perform unit tests on classes whose objects would usually rely on graphical rendering methods to run. This is done by calling the graphical library as a “mock object” with which a JUnit test can interact, as the code will not require the graphical library in order to run since it is a Headless Application.

Unit tests are not wholly reflective of how the software will work in practice, and some bugs only crop up when the different components of the system work together. Due to the limitations of White Box testing, we also planned to run System tests for the Black Box testing part of the process, as this provides another, more realistic view of the system under test from the user's perspective, and makes sure that the system requirements have been met. However, System tests are quite complex and we had short time constraints. Since the application does not render graphics during Unit testing, we took on a static inspection approach, performing manual visual testing with video evidence for the aspects of the game that rely on having a GUI. Manual tests also helped us assess the usability of our system. The combination of unit testing and manual testing provides a holistic view of the robustness of the project, and we can confidently conclude a full system test will have also been passed.
Benji and Joe will be responsible for writing test cases.

Brief Testing Report

Our testing was a success and led us to some great results. We ran a mix of both Unit and Manual tests with 28 Unit Tests and 21 Manual Tests. Overall we had a 100% success rate with our tests.



Comparison of Number of Unit Tests to Manual Tests

The Unit Tests were implemented efficiently and allowed us to enrich our Continuous Integration. These tests could be run using Gradle and then on completion gave us detailed information of how each test ran in a readable format.

28	0	0	0.022s
tests	failures	ignored	duration

100% successful

Classes

Class	Tests	Failures	Ignored	Duration	Success rate
BoatTest	5	0	0	0.005s	100%
GameplayTest	6	0	0	0.001s	100%
LaneTest	3	0	0	0s	100%
ObstacleTest	2	0	0	0s	100%
PowerUpTest	3	0	0	0.001s	100%
SaveLoadTest	8	0	0	0.015s	100%
TupleTest	1	0	0	0s	100%

Gradle Testing Report

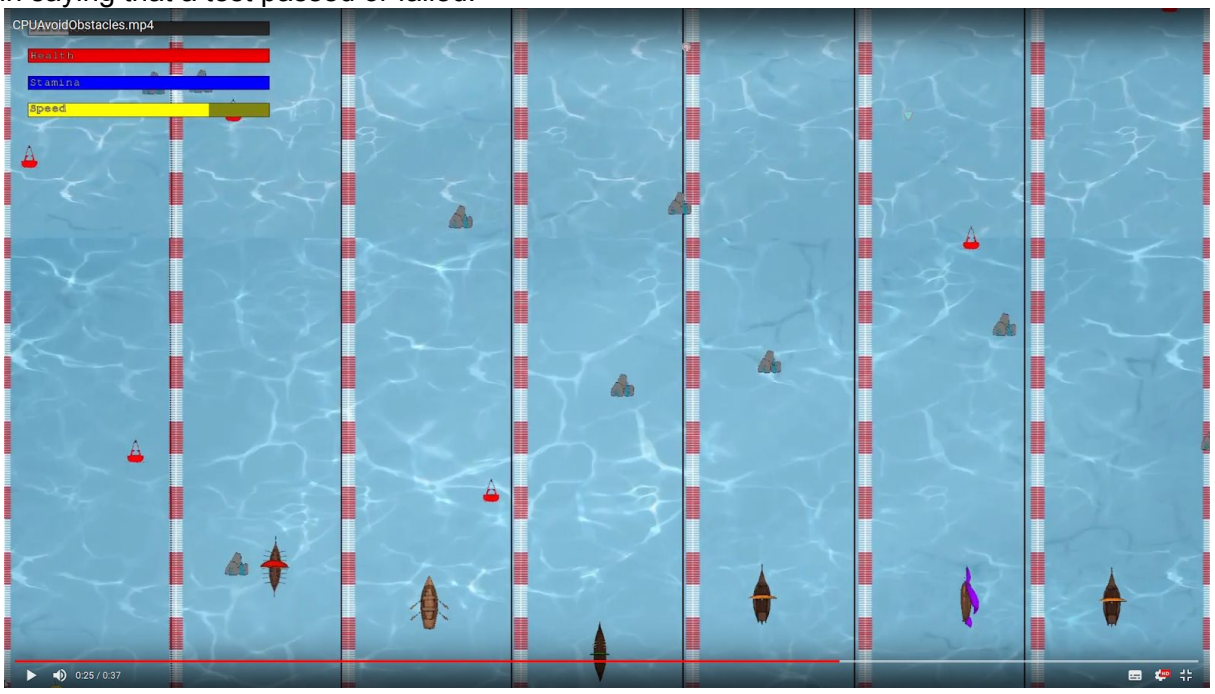
The 100% success of our unit tests led us to feel confident in the non-graphical backend of the code which consisted mainly of the mathematical calculations and organisation. This is highly important as any small faults in calculations could lead to large changes that could cause the game to not function correctly. When accounting for the Classes that could not be Unit Tested we achieved an overall Class Coverage of 97.7%, Method Coverage of 70% and Line Coverage of 60%.

We were unable to reach 100% line coverage, however we know 100% is too expensive to achieve. Additionally, 100% code coverage does not guarantee that our code is completely free of errors, nor is it a reliable indicator of the effectiveness of the test suite.

All unit tests were checked by multiple members of the team so that any small errors would be removed. This meant that all tests were actually testing the correct aspects of the code and small errors didn't throw our results. This led us to believe that our unit tests have a high level of correctness.

We also ran manual tests which were backed up with video evidence. These tests also had a 100% success rate. These tests were written to supplement the Unit Tests, as with just the Unit Tests alone there would be large sections of the code that would have gone untested. Each test tested a specific part of the UI or backend that required rendering to test leading to us having specific tests that are effective at allowing us to be more precise in our testing.

A video was taken alongside each test which allowed us to have multiple members of the team watch over each video and agree that each test was conducted fairly and was thorough enough that we could be confident in saying that a test passed or failed.



Example Manual Testing Video

As no manual tests failed we are confident in saying that our tests have a high level of correctness. Each test was supported with a rationale explaining why the test had been conducted in such a way and contained detailed instructions as to how to reproduce them.

We also created a traceability matrix representing the relationship between requirements and test cases to make sure that test cases that were created did not overlook testing some requirements, nor were redundant (i.e we didn't have multiple test cases testing the same requirements multiple times). They also allow us to know which test cases to change if requirements change.

Overall due to the large code coverage that our mixture of Unit and Manual tests investigated we have a high level of completeness, and we feel that each and every aspect of the code was tested thoroughly and to a high standard. The overall 100% successful results of these tests also shows that we have well written code that was able to stand up to vigorous testing. We feel that if it was to be tested by multiple users they would be unable to find any significant bugs which we did not catch. However, we realise our testing limitations in that

although we have tested our program to such an extent and with such success, we cannot definitively prove the absence of bugs in our program.

Below is both our full testing tables for Unit and Manual testing, with direct links to evidence, and also an easily navigable HTML file for a more visual representation of the Unit Tests conducted:

Unit Test Table: https://kingzoszn.github.io/files/UnitTests.pdf

Manual Test Table: https://kingzoszn.github.io/files/ManualTests.pdf

Unit Test HTML Report: https://kingzoszn.github.io/files/test-report/

Traceability Matrix: https://kingzoszn.github.io/files/TracabilityMatrix.pdf

GitHub folder containing all test files/designs:
https://github.com/Fluxticks/ENG1-DragonBoatRace/tree/master/tests/src/com/dragonboatrace/game

Bibliography

[1] ”Unit Testing” [Online] https://theqalead.com/topics/unit-testing/ [Accessed 8 February 2021].
[2] “Integration Testing: A Complete Overview” [Online] https://www.testingxperts.com/blog/integration-testing [Accessed 8 February 2021].
[3] Grill, T., 2015. *TomGrill/gdx-testing*. [online] GitHub. Available at: https://github.com/TomGrill/gdx-testing [Accessed 8 February 2021].