ENG1 Team Project
Assessment 2

Deliverable 4: Test Report

Team 13 – Team Unlucky

LILLY BROWN
ROSCOE GILLATT
ADAM JOHNSON
YUMIS ZYUTYU
BRANDON WEST
AYMAN ZAHIR

a) Methods and approaches

Testing was done in 2 different ways. The first was with a manual testing approach. This involved going through the actual game and testing features, functionality and some fringe cases. The reason we did this was twofold: for one, this allowed us to see exactly how some of these scenarios would occur in a live game environment, where some unexpected interactions could occur with other components of the game. The other reason is that some things are impossible to test in an automated testing environment, like visual bugs and user interface experience.

Our other methodology is Unit testing which involves coding tests to check value interactions and in some cases physics of objects for example. There are a few other benefits that Unit testing methods provide. The tests can be rerun automatically when a new version of the game is made (i.e. when a new .JAR file is generated) saving time by manually retesting some portions of the game to ensure no new bugs are found or old things are broken. Unit tests also allow us to check code coverage of the game, which effectively shows us how much of the game has been tested using unit tests. Coding environments (Such as IntelliJ) can be set up to check this every time a set of tests is run and only to check certain classes or directories. Testing coverage does not necessarily have to be 100%, as some components are either not testable, have no reason to be tested (such as simple getters or setters) or have simply been tested manually in such a way that automatically testing it would be redundant.

The way we approached testing was to start with the base version of the game provided by Team 3. Two team members were initially designated to work on testing whilst new features were being implemented. This was done to help make the use of time in an efficient way. We started developing tests for portions of the codebase that we knew would not change, along with pre-emptively testing basic features that were already present manually. This was to expedite the testing process for the final game by saving time on the test writeup.

Manual tests were recorded in a table. Each entry consisted of a unique ID, some basic information about the test (related components, test owner), what the test was actually for along with the required inputs, expected and final results and finally whether or not the test was considered a pass or fail. There were three categories for this:

- Pass, where the test concluded as we expected it to
- Fail, where the test concluded in a way which could not pass the test at all
- Partial pass, where the basic benchmark of the test is passed but with some unexpected effect also occurring.

Fails usually occurred due to related components not being implemented or not functioning at all.

Unit tests were set up as part of a continuous integration environment, which allowed them to run each time the game was updated. A test report was automatically generated in an HTML format, which can then be shared on the website for viewing. Code coverage can also be tracked by running it in an IDE.

All tests were grouped up based on relation and placed into a traceability matrix [https://apj520.github.io/ENG1Team13.github.io/pdfs/TMatrix.pdf] that was used to denote which requirement was tested by what test. This was to ensure all tests had a purpose and no requirement lacked some sort of benchmark to back it up.

b) Test report

General

There were a total of 91 Manual tests conducted, covering all aspects of the game. There were basic functionality tests that involved checking for functionality of basic features, Extreme case scenarios where impossible values were used to stress test the game and see if there were any unintended side effects and some usability tests to ensure all user requirements were met for aspects such as colourblind friendliness.

Unit testing was conducted, but was not as extensive as we would have liked. This was due to some difficulties with getting other areas of the game to work delaying the testing process, along with hardships involving writing the tests themselves. For what is available, there are 41 Unit tests, consisting of 19% of the lines in the code. This 19% ends up covering 35% of the classes in the project and 13% of the conditional statements present. For full details of the unit testing please consult [https://apj520.github.io/ENG1Team13.github.io/Test%20report%20(1).zip], but for an overview of all testing please see below:

Utilities

Utilities is a series of backend functions which provide useful calculations that can be used throughout the game. These were entirely unit tested as they were mostly mathematical calculations that would be impossible to see in a live environment. Code coverage for this was 92%, with the remaining 8% consisting of simple getters or setters. Most of the tests passed, with a few exceptions. This was possibly due to decimal errors, with the actual tests resulting in the correct values.

Movement

This was tested manually initially, where all the tests passed as expected. There are 8 different possible directions which the ship can be moved in, which can be stringed in combination. Collisions also work, where the boat is not capable of going anywhere that is not water.

Weapons systems

Weapons systems refers to both the functionality of the actual weapon (ie the cannonball) and also the related ammo system. This was tested using a combination of both manual and JUnit testing. A bug with a redundant firing system implemented by team 3 was discovered, so this was removed. Beyond this, basic weapon firing functioned as expected.

White box integration testing was used in order to test the ammo functionality e.g. firing shots reducing ammo and also testing to make sure the shots did not fire without ammo. This was done through setting the ammo value and testing to see if running the fire function reduces the ammo by the appropriate amount or doesn't reduce ammo Implying the shot did not fire.

Powerups

After several iterations power ups were tested both manually and via the use of unit tests. Earlier manual testing uncovered several bugs/glitches with this feature such as the doubleshot powerup intermittently turning on and off amongst other issues. By the end of development, all power ups functioned as expected even if their potency varied heavily in game.

White box integration testing was used in order to test the functionality of the double plunder and immunity power ups by activating them on the player and testing if the totals for plunder and health change appropriately. The double plunder power up was tested by adding 100 plunder to the player with double plunder activated and thus the player now has 200 plunder. The Immunity power up was tested by attempting to damage the player whilst the immunity powerup is activated and thus the player's health was not reduced

Shop

This has been manually tested and would seem to be impossible to test using unit tests. the buttons function correctly, and as long as the player has the plunder to spend in the shop they can buy items over and over.

Menus

This has been manually tested and would seem to be impossible and redundant to test using unit tests. All menu functions work properly, with the main menu mostly encompassing start functions, the esc menu encompassing leaving/restarting the game and the save button being integrated into the game UI. All buttons performed exactly as expected and passed all tests set of them.

UI

In game UI has 3 boxes: controls, objectives and the shop. The controls box acts as a basic tutorial and has no functionality to test beyond simply making sure it appears. The objectives outline what needs to be done by the player to progress the game and functions as expected, with completing an objective progressing it and skipping over objectives that may have already been completed by accident. The final UI component is outlined in the shop section. As this is either dependent on nothing but displaying it in game or heavily dependent on whole game features to test this was manually tested and passed all tests.

Map components

Tests were done manually, as these components of the game are heavily dependent on other parts of the game, such as map generation to be properly tested. The 3 tests did pass with notable quirks for picking up chests. This did not adversely affect gameplay as they were not prevented from being picked up but given more time this may have been changed. Several player affecting items are also loaded on the map, such as powerups and a boss Monster, all of which do load in correctly.

Quest

A white box integration test was used to test the chest pickup functionality. In this test a chest is loaded with the player in the pickup radius and is checked for quest completion which returns true showing that the player will be able to complete a pickup quest.

Combat

This was tested manually due to the time constraint brought on by how late this feature was implemented. Combat comes in one of 3 forms: Fighting boats, destroying colleges and “fighting” the boss. Boat combat is gone over in more detail in the AI section but all tests passed. Fighting colleges was mostly functioning correctly with the small caveat that destruction persists after restarts. This does have a small effect on the game but if given more time would be resolved. The boss acts as an obstacle but can be treated as essentially an indestructible immobile boat, which can harm the player. Most combat features (with the exception of the aforementioned restarting bug) passed their respective tests.

White box testing was used to test the player’s health in relation to damage/death. 3 tests were performed, one where the player has 2 health and takes 1 damage and thus does not die, another where the player has 1 health and takes 1 damage and thus dies and a final one where the player has one health and takes two damage and thus dies. These tests cover each side of the boundary of the dying functionality.

AI

Boat AI has several different complicated interactions which necessitated manual testing rather than unit tests. Even if this hurdle had been overcome due to how late the feature was implemented there was not enough time to write unit tests for this feature anyway. Boats have 3 main functions: Follow the player, fight the player if they are an enemy boat and fight for the player after they have allied with them. A boat can be allied by essentially defeating them in combat. All manual tests conducted in regards to boat AI passed with no notable bugs or glitches.

Extreme value testing

These tests were conducted to ensure functionality in technically impossible scenarios under normal circumstances. These were performed by altering a save file and then using this to load an altered game state. 2 types of test were conducted: one with extremely large values (10000000000) and the other with negative values. The objective here was simply to ensure the game didn't crash or become unplayable, which in that regard the tests succeeded. It did point out some interesting side effects however which given more time may have either been patched out or possibly even be integrated as possible features, such as infinite cannonballs provided by negative ammo.

Other

These were tests that didn't fit into more technical categories, and were primarily usability tests for some user requirements. This included checking for flashing, colour blindness friendliness and distance viewing experiences. Due to this this was impossible to test automatically, although after manually checking this these tests were successful.

c) Test evidence URLS

https://apj520.github.io/ENG1Team13.github.io/pdfs/Manual%20testing%20table.pdf

https://drive.google.com/file/d/11PqaGablnqefspZTlq8dp6OnzNnTIQ6g/view?usp=sharing

https://apj520.github.io/ENG1Team13.github.io/pdfs/TMatrix.pdf

https://apj520.github.io/ENG1Team13.github.io/pdfs/Automatic%20testing%20table.pdf

https://apj520.github.io/ENG1Team13.github.io/Test%20report%20(1).zip