Testing Report and Evidencing

Testing method(s) and approach(es)

Due to having used IEEE standards previously and successfully in our project we decided to use IEEE Software Test Management Process[1]. The model illustrates the layers of testing a piece of software in order to create an adequate report on said software. The standard uses a three-layered model that covers: Organisational Test Specifications, Test Management and Dynamic Testing. It allows for prioritised testing by using a risk-based approach so that the most important aspects of a system can be tested with a higher priority which can be useful if there is a time restraint on testing a piece of software. This was a contributing factor when selecting our testing method due to the tight deadline we have to appease in this project. The main reason that we chose this model is due to the fact it is very adaptable to any software testing. We felt that we could take the adaptivity of the model and really tailor it to our project. Following the IEEE Software Test Management Process, we found that the testing standard goes into a high level of detail on many different testing techniques. Due to the time restriction on the project, we modified the standard and selected techniques which we felt suited the stage which we were at currently in the project.

The aspects we have taken from the IEEE testing specification are the dynamic testing, looking at the functionality of the software once it has been compiled, and the presentation methods given (testing matrix) which can be viewed for our testing matrix in the additional documents section of our website.

First of all, we performed black box testing on the compiled game. We felt that this would quickly flag up any areas which, if failing, may be of interest to the development team. As the tests were run looking at the requirements specification mainly, not concerning how the internals of the code and game worked, we could very quickly find requirements that were not met or functionality in the game which was not working correctly. Due to our flat hierarchy, it was difficult to select a team member to run the tests, as the individual should have no knowledge of the inner workings of the game or structure of the code. We selected the design manager as the most appropriate member to run the tests as we felt that they were least acquainted with the code itself, and much more concerned with the requirements design and functionality side of the project.

Once the black box tests were completed, the results were placed in a modified testing matrix. These results were then analysed by the development team (who have a good understanding of the code). Due to the nature of the dynamic black box tests being time efficient, this quickly gave the developers information on where bugs may be occurring.

Our second testing method involved unit tests. To do this, we isolated different functions in the code and testing if they gave the correct output for both common test case inputs, as well as edge case inputs. This was carried out by members of the group who had a very detailed understanding of the code structure. If there were any bugs or failures present in black box testing, these would have been very quick to track down and isolate in unit tests which was our motive for black box testing first.

Once we tested isolated units of code, we combined all the code into the full program and ran the same tests to ensure that the results were not affected once the units were not running in isolation. After the tests passed and the bugs were resolved, the development team could then review the code making it more efficient in areas as well as making the code more readable and well documented.

Black Box Results and Statistics

We ran a total of 29 black box tests. All of these tests were carried out by the design manager to keep the tests true to the standard. Of the 29 tests, 4 failed and 25 passed. This showed initially that our game was very robust and had been implemented well. The reason for the small number of failed tests was largely down to ongoing testing which had been done throughout the development of the game by the development team as they were writing the code.

The tests that failed are listed below with reasons for failure (referencing from black box testing document):

- **1.26 -** This test was testing the requirement that the game would last between 30 and 60 minutes on average. The test failed due to lack of map size and lack of ships (both under user control and enemy ships under AI control). We know this will easily be affected by how much we scale up the map and fleet size (number of ships present in the game) and it is something that will be resolved when balancing the game appropriately.
- **1.28 -** This test was testing the randomly generated events and features that would occur around the map throughout the playtime of the game. The reason the test failed was due to the fact this requirement has not been implemented at this stage of the development process.
- **1.30 -** This test was testing the requirement of having a minimap in the corner of the screen to show the user the location of all their ships. The reason the test failed was due to lack of implementation at this stage of the development process. The reason the feature has not been implemented is due to the fact that we decided it was not needed. At this iteration the user can clearly see the whole map present on their screen at all times so during a group discussion we decided that this would be a useless feature to implement at this stage.
- **1.31 -** This test was testing the requirement concerning a ‘fog of war’ which restricts the user's vision. The reason that this test failed was due to lack of implementation of this feature. We felt that implementing the feature at this stage would make it difficult to test other aspects and requirements of the game.

Unit Tests and Results

Learning from the testing carried out in assessment 2 where supposed incompatibilities were found between LibGDX and JUnit, research was carried out and a solution was found. As LibGDX needed to be initialised before any its resources such as Textures could be used, JUnit would not work on methods that depended on said resources. The mocking framework Mockito was therefore used to remove the dependencies on LibGDX and create 'mocks' of various LibGDX dependent classes when needed. This resulted in unit tests which were now able to run without any of the messiness of manually inserting test code into the specified methods. The unit tests were now accurate and reliable, being focused on the logic of the method it was testing and not on the implementation of various LibGDX resources.
Of the 29 unit tests we ran, there were no failures which ensured that the code we had written had no logic errors causing any bugs.

Class: *Department*

ID	Test	Description	Result
1.1.1	*buyUpgradeNoGoldTest*	Test that player cannot buy an upgrade if they cannot afford it	Pass
1.1.2	*buyUpgradeDefenceTest*	Test that a department specialising in defence will only offer defence upgrades for the player's ship	Pass
1.1.3	*buyUpgradeAttackTest*	Test that a department specialising in attack will only offer attack upgrades for the player's ship	Pass
1.1.4	*buyUpgradeAccuracyTest*	Test that a department specialising in accuracy will only offer accuracy upgrades for the player's ship	Pass
1.2	*buyWeaponTest*	Verify that the player can only a buy a weapon if they have enough gold and that the right weapon is added to their owned attacks	Pass
1.3.1	*getUpgradeCostNoUpgradeTest*	Test that the player will not be charged if the department has no upgrade to offer currently	Pass
1.3.2	*getUpgradeCostDefenceTest*	Test that the correct cost is given for the defence upgrade	Pass
1.3.3	*getUpgradeCostAttackTest*	Test that the correct cost is given for the attack upgrade	Pass

1.3.4	*getUpgradeCostAccuracy*	Test that the correct cost is given for the accuracy upgrade	Pass

Class: *Player*

ID	Test	Description	Result
1.1	*payGoldNoGoldTest*	Test that a transaction is not carried out if the player has no gold	Pass
1.2	*payGoldFailedTest*	Test that a transaction is not carried out if the player cannot afford it and that negative value are not valid as the price for a transaction	Pass
1.3	*payGoldSuccessfulTest*	Test that transactions are carried out if the player can afford them and that their amount of gold is reduced accordingly	Pass

Class: *Ship*

ID	Test	Description	Result
1.1.1	*damageBroadsideTest*	Test that the damage given out by the broadside cannons is correct and distributed correctly across the ship's sails and hull	Pass
1.1.2	*damageDoubleShotTest*	Test that the damage given out by the double shot is correct and distributed correctly across the ship's sails and hull	Pass
1.1.3	*damageExplosiveShellTest*	Test that the damage given out by the explosive shell is correct and distributed correctly across the ship's sails and hull	Pass
1.1.4	*damageGrapeShotTest*	Test that the damage given out by the grape shot is correct and only applied to the sails	Pass
1.1.5	*damageRamTest*	Test that the damage given out by the ram is correct and only applied to the hull	Pass

1.1.6	*damageBoardTest*	Test that the damage given out by the board attack is correct and only applied to the hull	Pass
1.1.7	*damageSwivelTest*	Test that the damage given out by the swivel cannons is correct and only applied to the hull	Pass
1.2.1	*healSailsTest*	Test that the sails are healed to the correct value and not exceed the maximum health	Pass
1.2.2	*healHullTest*	Test that the hull is healed to the correct value and not exceed the maximum health	Pass

Class: *MiniGamePlayer*

ID	Test	Description	Result
1.1	*isDeadTest*	Test that it is correctly identified when the player has been killed	Pass
1.2	*movableTest*	Test that it is correctly determined whether or not the player can move in each direction and also if they are in the winning square	Pass

Class: *Attack*

ID	Test	Description	Result
1.1	*doAttackTest*	Test that the method correctly determines the amount of damage to be applied	Pass

Class: *DoubleShot*

ID	Test	Description	Result
1.1	*doAttackTest*	Test that the overridden method (inherited from Attack) correctly determines the amount of damage to be applied when Double Shot is used	Pass

Class: *ExplosiveShell*

ID	Test	Description	Result
1.1	*doAttackMissedTest*	Test that the overridden method (inherited from Attack) applies damage only to the attacking ship if their attack misses	Pass
1.2	*doAttackSuccessfulTest*	Test that the overridden method (inherited from Attack) applies damage only to the defending ship if the attack is successful	Pass

Class: *GrapeShot*

ID	Test	Description	Result
1.1	*doAttackTest*	Test that the overridden method (inherited from Attack) correctly determines the amount of damage to be applied when Grape Shot is used	Pass

Class: *Ram*

ID	Test	Description	Result
1.1	*doAttackTest*	Test that the overridden method (inherited from Attack) correctly determines the amount of damage to be applied when Ram is used	Pass