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.

Testing Report and Statistics

As discussed previously in the document, we first used black box testing, then we used unit testing. The full set of results for these tests can be found in the URLs listed at the of the document.

Black Box Results and Statistics

We ran a total of 32 black box tests. All of these tests were carried out by the design manager to keep the tests true to the standard. Of the 32 tests, 4 failed and 28 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.

White Box Testing

A total of 23 tests were carried out to verify the functionality of individual methods and their interactions with each other. They were categorised by which class they were testing. 22 out of the 23 tests passed the first run with the one failure being the program's inability to find the file in which the map of the game was stored. A workaround was promptly developed which then passed the test on the second run. JUnit was the framework used to implement these tests and was troublesome at times thus requiring some tests to be done manually. Issues included JUnit not being recognised by IntelliJ and having to be re-imported repeatedly, thus giving inconsistent results due to it not being synchronised with the rest of the program. However, these problems did not affect the efficacy of the tests themselves and we were able to get consistent and accurate results from implementing them manually by inserting the test code where it was required to give an output. The high pass rate is also helped by the fact that testing was done regularly throughout the development process and thus was used to guide it and maintain functionality while changes were made to the program.

Testing Material URLs

Black Box Testing - http://limewire.me/docs//assessment2/BlackBoxTesting.pdf
Testing Traceability Matrix - http://limewire.me/docs//assessment2/TraceabilityMatrix.pdf
White Box Testing Evidence -http://limewire.me/docs//assessment2/WhiteBoxTesting.pdf
Unit Test Scripts - http://limewire.me/docs//assessment2/unit_tests.zip
Post Test Bug Correction - http://limewire.me/docs//assessment2/BugCorrection.pdf

Bibliography

1 - ISO/IEC/IEEE 29119 Software Testing
http://softwaretestingstandard.org/part2.php[1]
2 - White box testing methods and description
http://softwaretestingfundamentals.com/white-box-testing/