Evaluation

When evaluating how well our product has met the brief provided, we made sure that all necessary requirements given by the client and written in the brief were satisfied. We also added extra requirements which we felt would aid the mandatory requirements set by the client. Implementing these additional requirements had much less priority than the mandatory requirements. We modified the requirements table from assessment 3, as the old requirements remained throughout all assessment iterations, and added in our new set of requirements. We also gave the new requirements a priority and the higher up the table, the higher the priority.

When evaluating the product we felt that, due to the way the requirements table and change log was presented, it was very easy to track which requirements had been implemented as well as the ones which we didn't meet.

There was an additional requirements brief provided to us before assessment 4. This required us to add in a new set of mandatory requirements provided by the client. The requirements document http://limewire.me/docs//assessment4/UpdatedReq4.pdf **[1]** was updated for this assessment. When adding new requirements or inferring from the mandatory requirements given in the brief, we made sure we had in-depth discussions with the whole team, as well as reviews of the provided brief, about ensuring that the requirements aided the storyline of the game. We also discussed how well the requirements improved the completeness of the game and we ensured that the added requirements aided gameplay and didn't affect how challenging the core gameplay was. If the new requirements we implemented made the game too difficult then the game wouldn't be enjoyable to play, and if they made the game too easy then the sense of achievement upon completing the game would have been taken away. Due to the game following a natural progression in the sense that there is motivation to capture Derwent, James, Vanbrugh, Goodricke to make capturing the objective college Langwith, we did not want to make the game too easy as this would have taken away the motivation to capture the initial 4 colleges before capturing the final college, therefore subtracting from the storyline of the game, as these colleges buff the ship.

We managed to extract and infer multiple new requirements from the brief. Addition of natural obstacle and new crew member, multiple natural obstacles. We felt that the brief was concrete enough for us to extract a solid set of requirements and we didn't find a problem with ambiguity. We did, however, feel like it was necessary to discuss the new requirements with the client in labs. During this discussion, we also talked about changing the previous group's minigame and the justification for doing this.

We ensured that all the requirements which we extracted from the brief had the highest priority for implementation. Once we had implemented all the necessary requirements as well as some additional requirements we felt aided gameplay and enjoyment, such as a full-screen map feature, we then tested them. This was mainly done using the black box testing method. Results can be seen here: http://limewire.me/docs//assessment4/BlackBox4.pdf **[2]**. Discussion of the results of these black box tests are discussed later in this document. The functionality of these requirements was also assessed using fit criterion seen in the requirements document.

Testing

To test the final product, we evaluated the quality of the software. We did research into what defines the quality of a product. We decided to use the characteristics listed in "Quantitative Evaluation of Software Quality" by B. W. Boehm et al. **[3]**. The literature defined a set of 11 software characteristics: (1) Understandability, (2) Completeness, (3) Conciseness, (4) Portability, (5) Consistency, (6) Maintainability, (7) Testability, (8) Usability, (9) Reliability, (10) Structuredness, (11) Efficiency. The definitions for the characteristics are written in the appendix of the literature. We decided to use all of these characteristics to test the quality of our software.

Characteristic	**Definition**	**Evaluation**
Understandability	The purpose of the product is clear to the evaluator [3]	➢ Code was consistent with appropriately named variables, methods and classes. ➢ Comments and appropriate docstring for non-trivial methods and lines of code ➢ Clearly written, simple code
Completeness	all of its parts are present and each of its parts are fully developed [3]	➢ No lines of code written without use later in the software ➢ No missing parameter values ➢ References to external libraries are all resolved and synchronised using Gradle
Conciseness	no excessive information is present [3]	Code written object oriented programming paradigm Documentation written concisely with clarity in mind. We have tried not to include redundant information in both code and documentation.
Portability	It can be operated easily and well on computer configurations other than its current one [3]	The software has low graphical demand and runs as an executable JAR file so can be run on any configuration that has Java installed.
Consistency	Internal Consistency - uniform notation, terminology and symbology within itself" External Consistency - content is traceable to requirements [3]	Internal consistency achieved, justification discussed in understandability. External consistency achieved. All documents regarding testing and requirements have appropriate IDs assigned to respective field.
Maintainability	it facilitates updating to satisfy new requirements [3]	Updates to satisfy new requirements shown clearly in the black box testing file and the software. All changes and updates evidenced and discussed in respective documentation.
Testability	Facilitates the establishment of acceptance criteria and supports evaluation of its performance [3]	Code tested with box black box and unit tests, evidenced later in this document. All unit tests are supported by comments and explanations in the unit test document.
Usability	Code possesses the characteristic of usability to the extent that it is reliable, efficient, and human engineered. [3]	User manual provided to allow any audience to understand how to play the game. Clear UI and logical progression throughout the game.
Reliability	It can be expected to perform its intended functions satisfiably [3]	High priority requirements met. Executable file has been demo ran throughout the course of the assessments with frequent playthroughs ensuring that the game doesn't crash or experience bug. Executable file runs consistently.
Structuredness	Code possesses the characteristic of structuredness to the extent that it possesses a definite pattern of organization of its interdependent parts. [3]	Code organised into appropriate classes with modularity in mind. Self contained methods. Clear structure to code with object oriented approach.
Efficiency	Code possesses the characteristic of efficiency to the extent that it fulfills its purpose without waste of resources. [3]	Methods written with efficiency in mind. LibGDX methods handles optimisation well.

We used unit testing to test the efficiency of the code itself, continuing with our approach from the previous assessments as it gave us an easy, partially automated way of seeing whether or not our code was acting as we expected it to. JUnit and Mockito were used again to create these tests. The unit tests themselves were written using JUnit however, when running these tests, any calls to LibGDX objects such as Textures or Screens would result in errors as LibGDX is not initialised when JUnit tests are run. To get around this in the previous assessments we combined the LibGDX headless backend with the Mockito mocking framework. This allowed us to 'mock' any LibGDX objects and run the tests without relying on initialisation of LibGDX, removing any external dependencies from the tests and giving us freedom to focus on the logic of the code itself without worrying about the implementation of the framework surrounding it.
Due to the similarity between the current game and the game we working on in the previous assessment, many of our unit tests were easily transferred across with some adjustment. New tests were therefore only required for the new mandatory requirements introduced this assessment and for extra features we implemented.

All 17 unit tests we created passed on the first run. Some tests are shown below, the rest can be found here:
http://limewire.me/docs//assessment4/UnitTestTable.pdf,
http://limewire.me.docs//assessment4/UnitTestScripts.zip (also available in the source code)

Class: *Department*

ID	**Test**	**Description**	**Result**
1.1.1	*purchaseNoGoldTest*	Test that player cannot buy an upgrade if they cannot afford it	Pass
1.1.2	*purchaseDefenceTest*	Test that a department specialising in defence will only offer defence upgrades for the player's ship	Pass
1.1.3	*purchaseAttackTest*	Test that a department specialising in attack will only offer attack upgrades for the player's ship	Pass
1.1.4	*purchaseAccuracyTest*	Test that a department specialising in accuracy will only offer accuracy upgrades for the player's ship	Pass
1.2.1	*getUpgradeCostNoUpgradeTest*	Test that the player will not be charged if the department has no upgrade to offer currently	Pass
1.2.2	*getUpgradeCostDefenceTest*	Test that the correct cost is given for the defence upgrade	Pass
1.2.3	*getUpgradeCostAttackTest*	Test that the correct cost is given for the attack upgrade	Pass
1.2.4	*getUpgradeCostAccuracy*	Test that the correct cost is given for the accuracy upgrade	Pass

Meeting Requirements

Once development of our software ended, we had a group meeting discussing how well we met the requirements **[1]**. We used both the gameplay of the product, to assess how well it followed a storyline and represented an abstraction of the university campus, as well as the black box tests for the requirements we could do this with (non-opinion based).

We found difficulty when assessing how well the software appeased non-functional requirements as these were largely based on opinion. This is why we felt that a large group discussion was required. During this final meeting, we all performed a playthrough of the game independently taking different actions on each playthrough. Any bugs or potential errors were to be noted down and later discussed. We then got friends and family members to play the game and at the end of their playthrough we asked them to fill in a questionnaire asking how well they felt the game appeased the non-functional requirements. This can be viewed here :**http://limewire.me/docs//assessment4/Questionnaire.pdf** . All candidates for the questionnaire were provided with the user manual and a brief description of the storyline of the game. The results of the questionnaire can be seen here: **http://limewire.me/docs//assessment4/QuestionnaireResults.pdf**. We feel that these results support our beliefs that our software meets the non-functional requirements.

Out of our 23 functional requirements defined, we could run black box tests on all but 2 of them. These 2 are opinion based and do not have a clear black and white pass/fail. These requirements are requirement 5 and 18. Out of the 38 black box tests, 34 passed and 4 failed. This means that the requirements F19, F22 and F23 were not met. These were additional requirements which we added as a group because we felt that it would aid gameplay.

The tests that failed are listed below:

- 1.31 - This test was testing the requirement concerning a ‘fog of war’ (F22) which restricts the user's vision. The reason that this test failed was due to the 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.
- 1.17 - This test was concerning the requirement about weather conditions (F23) which we did not feel like we had time to implement well especially with this assessment iteration being the final product which we wanted to be as polished as possible keeping it bug-free.
- 1.18 - This test was ensuring that the ship could not move through squares which were not water as this does not fit the behaviour of a ship in the real world. This failed due to a bug which occurred when resizing the window when moving towards a body of land. The ship, in turn, gets stuck inside the body of land, island or college. We looked into this issue extensively after picking the game up from the previous group as this is when we discovered the bug. After a group discussion, we decided to move forward with further implementation of other features as the frequency of the bug occurring is small and the issue lies with the way LibGdx interfaces with coordinates on a TileMap. To fix this bug it may need a full game reworking which would cost too much time on the tight schedule provided to us for assessment 4.
- 1.29 - This test was regarding the requirement concerning a save/load feature (F19). The test failed due to lack of implementation. We felt that due to the reasonably short play time for the game (30-60 minutes) it was not a massively high priority requirement. We did not implement this due to the tight time restriction on the assessment and as a group, we wanted to focus on making the game bug-free and making sure all implemented requirements worked concretely. This is where the majority of our time was spent. There is also a risk associated with implementing this feature so it could have caused problems in the final product.

Bibliography

[1] Limewire Requirement [Online] Available:
http://limewire.me/docs//assessment4/UpdatedReq4.pdf

[2] Limewire Black Box Tests [Online] Available:
http://limewire.me/docs//assessment4/BlackBox4.pdf

[3] B. W, Boehm , J. R. Brown, H. Kaspar, M. Lipow et al, Quantitative Evaluation of Software Quality, Oxford: North-Holland Publishing Company, 1978.
http://csse.usc.edu/TECHRPTS/1976/usccse76-501/usccse76-501.pdf