Evaluation

Link to previous requirements: PrevReq
Link to final requirements: FinalReq
Link to user survey: Post-Game Survey
Link to user survey results: Results

The game was evaluated to ensure met the brief by using the requirements from ‘pi-rates’ as well as the additional requirements given to us by the client. By looking through the previous groups requirements, we confirmed that they sufficiently covered the brief that was initially given by the client, however they had not completely some of the requirements due to time constraints - as detailed in the next section of this document. Therefore, the game was in a state that was sufficient but not complete - there were minor implementation details that needed to be altered. In order to evaluate the final project entirely, we assessed the additional requirements to allow us to compile a final requirements document that would cover the additional requests for this assessment as well as ensure no previous requirements had been missed. We reviewed our previous requirements to confirm that none of these new requirements conflicted, to prevent combination issues upon implementation.

We checked each individual existing requirement against the game to make sure it met them. We checked the fit criteria for each requirement and made sure as many as possible matched, most of which could be done by checking a certain feature is in the game and working. Some of requirements could be confirmed through testing such as “NF2.1 Crashing rate should be at most 5%.”, while others required a user survey to see if it had been sufficiently met. The requirements that needed a user survey to be tested were non-functional, and the survey we used is linked above. The reaction to the user survey was reasonably positive, the average enjoyment was 3.5/5, as people thought it was a ‘good game’ with a ‘logical progression’. However there were a few concerns with elements such as pricing, as respondents said they were prefer as a free game though maybe ‘with ads’. We were overall very pleased with the responses we received as people mostly enjoyed it and thought it was a strong basis for whatever is made from it; so we consider the requirements met.

To meet the new requirements for assessment 4, we first met as a group and came up with the specifics of some features that we could implement in order to meet all of the requirement changes. As we implemented it, we checked it against this initial idea. Once we finished implementing it, we went back and compared the completed game to our new requirements to see if they had been met by the mechanics we added.

Testing

For our code to be of appropriate quality:

- It should produce the expected value for all expected situations in gameplay.

- It should be easily readable with appropriate comments so that it can be understood by anyone who has to view or change the code.
- It should be easy to expand with new features or improvements.
- It should run in a short enough amount of time that the player doesn't notice any pauses.

To produce the expected values, the code should work as expected and each section should give the correct and consistent output for a given input and state. This can be confirmed by breaking the code down into smaller sections and using white box testing to confirm each small part produces the expected result for a given input. If it is assumed the results are passed between these sections correctly, if the individual sections work correctly, the whole thing can be expected to work correctly. It can also be tested with black box testing. The game can be run through most expected gameplay scenarios and be tested to confirm that the expected result occurs.

To be easily readable, the code should have a consistent layout and style, should have clear variable names, and should be appropriately annotated in sections that are not immediately obvious how they work. This can be solved with white box testing to read through the code. We initially wrote the code to these standards but, to confirm, a different person from the one who wrote the section should go through and confirm that it follows the coding style and is easy to understand.

To be easy to expand, the code should be structured in a clear and simple way. This can be tested with white box testing. Someone can go through the code and confirm that all concepts within the game are grouped and part of classes that make sense. By confirming that the comments are clear, they can also confirm that sections that may want to be expanded such as types of ship can be easily found.

The code can be tested to see if it runs well using black box testing. Someone can play the game and record any examples of long waits examples of frame rate drops or long waits. If none are found, the game can be said to run well. If some are found, the severity can be measured by timing the wait or recording the minimum frame rate and judge if they fall within acceptable values for the game.

The method of testing we decided on was to continue and expand on the white box and black box testing from the previous group. We felt this was an effective way of testing as it covered both the immediate user experience to check if anything is immediately or obviously wrong and broken, as well as checking all the code to check for any potential errors in code that can result in more rare but potentially more major errors when the game is run.

White Box Testing

There weren't many areas in the new sections of our code for this assessment where jUnit tests were appropriate. Instead, we went through and reviewed the jUnit tests from previous assessments to confirm they work and are sufficiently detailed. We also assigned different

people to read through each other's work to make sure it was up to standard in terms of style and readability.

For assessment 3 and 4, given that the graphical elements of the program could not be removed from the functionality, JUnit tests became very hard to implement given their lack of support for graphical elements. Therefore, we focused much more on what we could test by playing through the game in black box testing.

Black Box Testing

We went though and played the game, testing each input on each screen to confirm it has the required effect. We also came up with a collection of potential risk areas for errors such as trying to buy something you don't have enough gold for to see if these caused problems for the game.

For each requirement, we went through and made tests to see whether all the requirements were met by observing and playing through the game. We made several testing tables to this effect, which resulted in a tuple of 'Test No., Requirements Reference, Method, Predicted Result, Actual Result and Pass?' for each test. Which covered which requirement was being tested and what it was, how it was being tested, what we expected from the test, whether we got that result and whether the test passed.

For the final testing of the product in assessment 4, there wasn't much to do given that we were just building on previously made tests for assessment 3. However, we did make two new requirement tests to account for the changed specification, and for one of the requirements the previous group didn't meet. We felt that was sufficient for our testing purposes, given that there wasn't much additional content added for this section and that the previous aspects of the project had been very well documented and tested.

Meeting Requirements

All requirements are referenced by ID which can be found in the final requirements document referenced here. The functional requirements that were not met by the previous team were **F3.3**, **F8.1**, **F8.2**, **F8.5** and **NF2.3** - they have now been fully implemented except for **NF2.3**. Our final product fully meets the new requirement fully. The brief given meant 2 new requirements were created **F13** and **F14**, which were fully implemented. For specific implementation details please view the Impl4 documentation, where code and architecture changes have been well documented. Below is a table documenting the requirements needing to be implemented and how we have done so.

ID	Reasoning for Previously not implementing	How implemented
F3.3	Time constraints	Now we have typhoons which are random encounters and are obstacles to be avoided
F8.1	Didn't fit with direction of game	Not implemented. We still believe that such an addition would not fit the current game, and so we've kept to the idea that the player needs to first focus on guarding ships to receive enough gold to get the upgrades required for beating the college.
F8.2	Time constraints	Following on from F8.1, we decided not to implement the guarding ships. Instead attacking a college directly challenges a more difficult enemy without needing to battle the guarding ships.
F8.5	Time constraints	Now when you defeat a rival college, you receive an additional crew member. This comes along with an upgrade to either the player's total defence or their total attack.
NF2.3	Unnecessary	Not implemented. We agreed with the previous team that this requirement was unnecessary, as having a save/load feature seemed unreasonable for such a short game. As the game is intended to be used to demonstrate at open days, it seemed unnecessary to reload saved games since the target audience would only play for short bursts, where completion was not the aim.