Evaluation and Testing Report

We used several methods in the process of evaluating our software, both during development and after we finished. The first and most frequent was comparing our progress and updates to the new requirements that we created at the start of this part of the assessment (these can be found in the 'Final Requirements' document [1]). These requirements were used to inform our product backlog on Trello. When a task was completed we would discuss, in meetings, whether it had been completed to a satisfactory standard. whether the requirement it related to had been fulfilled and, if not, what work needed to be done to ensure the requirement was completed. We are comfortable to say that our game matches all the requirements that we set. We also discussed the completion of the new features with reference to our new use cases [2], which allowed us to evaluate the game by playing it and seeing if the experience matched what had been set out. Lastly, we introduced the game to others outside of our group during and after development. This helped us to gauge their comprehension of the finished game, which seemed to be reasonably high, with the majority of users intuitively understanding the new features and more largely, the game itself.

To ensure our code was of appropriate quality for full functionality of our game we used a mixture of tests. We took an approach to testing that was similar to the methods we had used in assessment three. This comprises of white box and black box testing as well as peer testing. Due to most of the game being GUI based, again we've had to test all GUI elements with black box testing.

What we mean by appropriate quality of code is that it is fully functional; the code makes sense to the reader in the sense that the variable and method names are appropriately named to what task they contribute to. All naming follows the convention for the language (Java/camel case) that is used. When it comes to the structure of the code, we consider modularity and as small amount of redundancy as possible. This allows easy reading and maintaining of the code. Lastly, good commenting of the code to give a description of the task being performed and any possibly confusing sections should be well explained but not every line is necessary to be commented.

In addition to this, the code in question should provide the functionality required to allow any relevant requirements to be met to the fullest extent that the existing code allows. That is to say that code is of an appropriate quality if the code that has actually been implemented (sections that have not been implemented cannot be judged on code quality) satisfies related requirements to an appropriate level.

Using black box testing on GUI allowed to check for good quality of code by simply seeing if the GUI met the requirements. This meant that nothing was out of place; everything showing on the screen whilst playing made good use of space and was appropriately placed. This also allowed to see if any GUI elements where not displaying as they were envisioned to i.e. there was flickering of some elements or if they didn't show at all.

White box testing allowed us to get a better understanding of if the code was actually doing the task it was designed for. Elements that we could create tests for within our project were the saving and loading of the game, the punishment cards and the new units. These tests create dummy data to test on. Each method that interacted with this data and didn't relay that data to affect a GUI element was tested. So for example the save and load features can mostly be tested using white box testing.

Peer testing occurred each time the game branches were merged into one branch. Each owner of a branch would explain the changes they made to the program as a whole to the colleague they were merging with and then vice versa. This allowed each person to thoroughly check the code was of an appropriate quality. As a merge happened, sometimes

conflicts occurred or not. Each conflict was discussed between the two parties until the produced code had the same effect on the program as each had separately allowing for both new features to be operational after the merge.

Each test created has been put onto our website [3]. This gives you a document that reports on each test we performed and what data was used/produced.

When we inherited this project there was just one unit test, this has been removed as it was testing the save and load feature of the project. Saving and loading has been heavily edited simply because there was a lot of redundant data being gathered and saved to file. Testing of the feature has now changed as all of the redundant data has been removed and the bare necessities are now being saved the features that were being tested have been removed. A new test has taken its place as described above, this tests the saving and loading with the new methods.

The save and load test passed all it's conditions. This has been tested with wrong and correct data alike. Although, as would be expected, the test didn't initially pass all of the conditions. This was down to minor misspellings as well as a few cases where there was not an appropriate manner to take care of occurring errors. The errors that occurred were the type that should always be taken care of i.e. the wrong type of variable being passed. Exception handlers have all been added as a result of the test to prevent it from happening again.

The GangMembers test also passed all of its conditions. This passed first time simply because they are not very detailed classes as they only contain information on the units strength.

With regards to black box testing [3], we ran a number of tests for both the Postgraduates system and the Punishment cards. In total, 27 black box tests were run, with all but 2 passing. Our approach to black box testing remained the same as the previous assessments, with test cases being written for each possible scenario that could both feasibly arrive and we deemed reasonable (likely enough to be able to both reliably test and to have a reason to test).

Completion of Requirements

This part of the document will cover the extent to which we've fulfilled the final requirements. Our requirements are split into four distinct categories Constraint, Functional, Non-Functional and Performance requirements. The completion of our requirements will be discussed in terms of the extent to which we've fulfilled these categories of requirements, using examples to demonstrate.

Our constraint requirements have been met to completion. The program can be run on a given computer in the computer science department *(C1)* and uses a keyboard and mouse as input for the game *(C3)*. After presenting the game to people inside and outside of our cohort and receiving positive feedback, we feel that we met our last constraint requirement *(C2)* which was to make the game appeal to our cohort and other prospect students. [REF]

With regards to the existing functional requirements, we believe that they have been to the best of our capabilities. In cases where we felt the requirements from assessment 3 were not properly met, we took time to improve upon the work that was handed to us so that the finished product would better meet all of the functional requirements. For example, *F3* (concerning the turn timer) was not met in the version of the project that was handed to us. We also felt that the save and load system was inadequate *F12*, so we made necessary

improvements. The majority of existing functional requirements had already been met when we received the project at the start of assessment 4.

The additions to the game outlined in the 'Requirements Changes' document created the need for the addition of a number of functional requirements. The existing requirements were abstract enough such that they didn't require changing. Instead, we considered it more appropriate to construct new functional requirements (*F19-F23* [1]). Since enacting the proposed changes to the game was the primary focus of this assessment, the majority of our time was spent making additions and changes to the code, such that the product reflected these new requirements. We feel that as a result of this attention, all of these requirements have been met and are confident in the quality of their implementation. For example the punishment card system required by *F21* has been implemented, surpassing the variety in effects and appearances required by *F23*. We confirmed this with reference to the use cases we generated [2].

We feel all Non-Functional requirements were met excluding NF5, which was removed as we did not feel we had the appropriate knowledge to implement accessibility features, or the time to effectively research and implement them, providing enough time for the other, higher priority, features to be implemented. For requirement NF1, we found that when having friends or other student play the game, they found the game easy to pick up; therefore meeting the requirement. Lastly, the single performance requirement (P1) has been fulfilled to the best of our abilities. We are confident in the quality of our code, and actually made significant improvements on the code that was given to us, eliminating a significant number of bugs. The game has run smoothly on a variety of systems, including the computers in the CS building's labs.

References

[1] "Final Requirements Updated". Sid Meier's. [Online]. Available: http://sidmeiers.me/documents/FinalRequirements.pdf [Accessed 1st May 2018].

[2] "Assessment 4 Use Cases". Sid Meier's. [Online]. Available: http://sidmeiers.me/documents/Assessment4UseCases.pdf [Accessed 1st May 2018].

[3] "Test Cases". Sid Meier's. [Online]. Available: http://sidmeiers.me/documents/TestCases4.pdf. [Accessed 1st May 2018].