Part A

Testing Methods and Approaches

Automated Testing

Before adding any new features, we first tested the base game by writing Unit Tests to ensure that the original (base) game covered all the initial requirements and had these tests in our Continuous Integration. From here, the team decided to implement an iterative development method where the team would plan and write write code for the new feature/requirement and then work on writing tests for the same, and would only move forward to another feature once all the tests for the previous feature had passed. By using this tactic, we ensured that the old features remained unaffected when new features were added to the code base. All the tests were created based on the requirements documentation and were aimed to cover most of the requirements, and we also considered the idea of code coverage and getting as much of it as possible as in headless mode, we can't test any elements which required user input (such as moving the ship by using keys), and therefore the code coverage does not reach 100%. All the automated tests were run when we push to the repository as well as on a pull request. This ensured that if any bug was found, the team would be notified and a member from the team could work on fixing this bug. All our tests are documented with relevant id, description, input, expected outcome and author. The code coverage reached is XX% of classes and XX% of lines. All our tests can be executed together by using the "test" Gradle task, or by using the run option which is offered by most IDEs for each test class. As described by Ian Sommerville [1] tests where created in such a way that individual object classes were tested, for example `collegeExists()` checks that the colleges are created alive. This would be unit testing. Other tests were created to check the components of a function, for example `collegeFires()`. Others test how some or all components integrate and test the system as a whole, for example `powerUpPickup()`.

Manual Testing

We implemented Black Box and white box testing during manual tests which we developed based on our tracebility matrix (features which we could not test automatically) as automating these was beyond the scope of the ENG-1 project. To create the tests, we referred to the requirements documentation and aimed to test as many requirements as possible. Our main aim was to first test the user requirements followed by testing the functional requirements. As we were slightly ahead of schedule, we decided to test the non-functional requirements as well. It was very important for us to test elements that utilized user input such as pressing keys/using the mouse in order to move the ship/attack other ships or colleges, as testing for user input with headless application is not possible in gdx-testing. We also manually tested the UI elements of the game as it wasn't possible to test them automatically. The teams' main goal of using this testing method was to ensure that the requirements were met, and any bugs were found and rectified. The white box element of the testing involved the implementation team suggesting ways that the tests could expose bugs in the game due to their knowledge of the code base. Majority of our test cases covered inputs which we thought were the best to test and ensure that all the requirements were met, with some additional tests covering invalid inputs, and some covering boundary inputs which would help in identifying any bugs. The traceability matrix as well as further evidence of testing can be found on our website.

Why?

By using the approach outlined for our automated tests, we believed that it was appropriate for our project as this ensured that throughout the development of the project, any features which were implemented in the previous (base) game as well as any new features which we were implementing were not unintentionally affected. Moreover, this approach satisfied our continuous integration, as the tests were run on every push to the repository as well as on any pull requests. The focus on black box approach for manual testing was suitable for such a project with a big codebase and with a certain amount of time. Using manual testing enabled us to easily keep track of the user requirements that were met by the game.

[1]I. Sommerville and M. Paul, Software engineering--ESEC ’93 : 4th European Software Engineering Conference, Garmisch-Partenkirchen, Germany, September 13-17, 1993 : proceedings. Berlin ; New York: Springer-Verlag, 1993.