**Testing Methods and Approaches:**

Although we are aware that testing will always have serious limitations - you can never test everything - the methods of testing listed below have been carefully chosen to check our game as effectively as possible given the finite time we have to choose from infinite possible tests. Testing was sometimes done in parallel with the code, however, we chose to follow test-driven development process as it seemed most appropriate to our project. Perhaps the most useful aspect, although daunting, was writing the tests before the code as it helps to focus on what the code being written should accomplish. It allowed us to find bugs early in development when tests failed and this faster feedback allowed for cleaner code and more time to complete other tasks. The cycle of failed test, passed test, refactor [1] proved invaluable as it ensured code did not just pass that specific test, but that it could be used to test the functionality further as the game took shape.

One of the ways that we implemented testing was through unit tests, using JUnit. Unit testing is exceedingly helpful for our model of AGILE development, as it allows easy refactoring of code as we can quickly check whether changes do or do not break the program. JUnit also is convenient for documenting code, as it allows programmers to easily tell what the purposes of different parts of code are. JUnit itself is good for unit tests, as it is easy to use as you can simply assert what should be the correct answer in different cases. It is also supported by default by IntelliJ, making setup and use simple.

Using integration testing proved essential as it allowed a number of units to be tested together which was invaluable in ensuring the code functioned correctly when tested as part of a larger project. This is therefore why we decided to use a ‘top-down’ approach where modules are integrated progressively after being tested from entry point, as explained in lecture 11. For this, we used a combination of black box and white box testing. Black box worked well with the test driven development approach as no access to the code would be needed. White box testing allowed us to design tests to follow branches and ensure conditions were checked.

System testing allowed us to test the functional and non-functional requirements of the game. It allowed us to evaluate how well the program complied with the requirements. As it tests the integrated system it allowed for errors we did not expect to be caught and therefore rectified. It seemed only appropriate to use black box testing for this.

Acceptance testing was the easiest to write tests for as it was specific to a particular scenario, however there are infinite scenarios so it was important to test only relevant scenarios and a wide variety of situations, as explained in lecture 11. Although the testing described thus far falls under the dynamic testing, static testing was used in the form of a walkthrough, as multiple coders risk lacking a cohesive approach to the project. This also ensured the commenting was clear enough to allow the project to be picked up by new coders.

[1]https://easternpeak.com/blog/a-test-driven-approach-to-app-development-the-main-benefits-for-your-business/

**Testing Report:**
Link to Questionnaire: Q
Link to test evidence: Unit and Function Tests
Link to Java test: Testing.java

**Testing Report:**

We split up the tests into two main sections, Requirements testing and Unit testing. Requirements testing went methodically through each of the updated requirements, creating a test for each one based on what was needed. This covered a mix of system and acceptance testing, depending on the requirement being tested and mainly focused on what happened on the user end of the experience. Whereas unit testing focused on making sure everything worked from a programming perspective, testing whatever wouldn't be obvious just by running the game. The split allowed us to focus on user experience with requirement testing, making sure the front end of the game worked well. With the backend being covered Unit Tests, aiding current developers with writing functional code and future developers with understanding code functionality. The requirements tests were designed up front and never changed, whereas the unit tests were gradually added in throughout development by one of the programmers when we had a good sense of where it would help and specifically what we were working towards.

**Requirements Tests:**

Our requirements tests were entirely based off the end requirements of the game, so as to give a higher end view as to whether our objectives when making the game were met. They were largely successful (29/35), as the only tests that didn't pass were ones that relied on functionality we haven't yet implemented. Of the tests we failed, all but one were failed by default given that the feature was still to be done in future. The one exception, would be the underwhelming length of the game (it took most players < 5 minutes to finish), but this can still be accounted for by the additional content that will be in the game in the future. To pass the tests in future that haven't yet passed, it is relatively straightforward as long as the rest of the game is completed successfully, as the rest of the requirements are built on that. We only had one test per requirement, as most of our requirements were very straightforward to test. We carried out the tests mainly through playing the game for the tests that warranted it, passing or failing tests when we managed to confirm that the game did or didn't do something properly. The only exceptions being the user experience requirements A4.1-A4.3, which were covered satisfactorily by a user survey. As well, as R35 which just required checking no assets were copied.

**Unit Tests:**

The unit tests were all successfully passed (24/24), and they all cover the most important aspects of the non-visual classes (the visual classes are covered indirectly through functional tests). We are confident that the unit tests and both complete and correct, correct in that they all ran and are reproducible by running the testing file again. They are complete in that test both the individual classes themselves and how the work, but also important

multi-class functionality that provides much of the main game, such as how taking damage interacts with the Ship, BattleMode and GameLogic classes.

**Evidence:**
The evidence for the functionality tests can easily be seen by playing the game itself, as they are able to be confirmed or disconfirmed by user experience. The unit testing evidence is comprised of ‘Testing.Java’ which was used to carry out the unit testing, and is linked to below here. The table consisting of the test results is also linked at the start of this section.