Testing Report

Stakeholders: Richard Paige, University of York Communications Office
Team: Barney Morgan, Cameron Smith, Harry Berge, Jake Phillips, Matthew Wilkie, Rob Weddell

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. Tests were done when something new was implemented to see if it met our expectations and at the end to make sure it still worked as it should do. 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 by giving us a clear end goal whilst also helping us structuring our code. 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. We used a variety of different testing methods to ensure our code worked as well as we hoped for, we opted to use peer testing, black box, white box, JUnit tests and requirements testing as we thought this would eliminate most, if not all errors. ~~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.~~

The most simple type of testing used in this project was peer testing, this is the first 'barrier' the code goes through before being implemented. Peer testing is making sure each member of our group checks over the changes to the code to see if they were happy with what has been done. Another advantage of this is each member can suggest improvements to the code if they are not 100% happy with it, making our code as efficient as possible. It has been shown that peer testing reduces between 85% to 90% [1]of bugs in the code so this is a very important method in any software project.

One other way 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[2] as we can quickly check whether changes do or do not break the program just by running it. 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. Most main aspects of our project can be tested this way making this a very valuable tool. ~~It is also supported by default by IntelliJ, making setup and use simple.~~ Although the JUnit tests can be difficult and lengthy the write the time saved once they are written massively outweighs this as they are used many times during the development process to test code in a matter of seconds.

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[3]. 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 and allowed members with little knowledge of java to participate. White box testing allowed us to design tests to follow branches and ensure conditions were checked. Integration testing meant we could test the game as a whole each time a new development was made instead of having to test each part individually each time which saved huge amounts of time which is critical when under a strict time frame.

Requirements testing isn't a different type of testing, istead it's used alongside other methods. We used this throughout each other type of tests completed. This makes sure that the client's specification was met by closely relating the tests to the requirements spec [4]. We created various tests to see if the requirements were implemented and if so making sure they actually work as expected. Most of the requirements were testing during blackbox, these tests were created using the list of our updated requirements list. [5]
~~System testing allowed us to test the functional and non functional requirements of the game. It allowed us to evaluate how well the program compiled 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:

The team used a wide variety of testing methods to give the best possible chance of eliminating errors by leaving nothing unchecked. We split the tests into each type of testing as mentioned above. The amount of testing meant that we came into a lot more errors than we expected but these were fixed to allow our project to run perfectly. Due to the requirements testing we are confident that the project meets the clients requirements, making this so far a success.

The overall testing process was really successful with only two tests failing the final testing process. Only visuals and minor changes need to be made to this further down the line for it to be a complete project but again these were not complete due to time constrictions.

White box testing:

In total 5 unit tests were created during development of our project, these tests aimed to test the most vital parts to ensure full correctness of these as these would cause major issues if they did not work. Each unit test passed. The main parts had to be unit tested as they had to be checked to see if new implementations did not cause any unwanted changes and due to the shear amount of times they had to be tested.

It was difficult for the team to create a unit test for each component as some required elements only available when the game is running (OpenGL related classes and internal files). Instead we made sure to stick to stringent coding standards and rigorously tested them using our various other methods to ensure no errors were missed and could be resolved before it was too late.

Black box testing:

Blackbox testing had the most test cases with 43 tests being carried out, most of these were based on the requirements to see if these had been fully met. From these 43 tests only 2 failed with one of them only failing due to the requirement no longer being applicable. The other failing test was with the battle mode as we felt like this had not been fully tweaked to perfection yet, we agreed this could be improved in a final implementation so decided that test was a fail. Overall these tests deemed our project a success as all the user requirements had been met so it's on course for what was expected. As we coded the project while relating to the requirements only one round of black box testing was carried out. For completeness the table of failed tests is given below:

Test ID	**Reason for failure**
23	The requirement that this related to was no longer applicable to our project.
43	The team did not have enough time to fully balance the game as intended.

Testing Evidence:

Link to Unit tests class

Link to Traceability Testing Matrix

~~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.~~

References

[1] ProfessionalQA - Peer testing [Online] Available: http://www.professionalqa.com/peer-testing

[2] Manifesto - Unit testing best practices for more effective code [Online] Available: https://manifesto.co.uk/unit-testing-best-practices-java/ [Accessed 15 Jan. 2019]

[3] SEPR Lecture 11 [Online] Available: https://vle.york.ac.uk/bbcswebdav/pid-2846241-dt-content-rid-7060327_2/xid-7060327_2 [Accessed 15 Jan. 2019]

[4] TutorialsPoint - Requirement based testing [Online]. Available: https://www.tutorialspoint.com/software_testing_dictionary/requirements_based_testing.htm [Accessed 15 Jan. 2019]

[5] Element of SEPRise!, Updated Requirements [Online]. Available: https://sepr4.github.io/web/submission/assessment3/updated/Req3.pdf [Accessed 17 Feb. 2019]