Software Testing

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

<u>Testing Methods</u>

Given that the IEEE standard was followed during the process of eliciting requirements, it was decided that the ISO/IEC/IEEE 29119 standard for software testing [1][2] was the natural choice. This testing method goes hand in hand with the team's requirements specification and ensures sufficient testing throughout the project. The team decided that time, precision and thoroughness were the most essential factors in choosing a method of testing, and it appeared that the IEEE standard fit them all. However, as the proposed system is relatively small and not critical, some aspects could be omitted without much impact on the thoroughness of the tests. It was found that the key aspects of testing were: black box, white box, peer, and requirement testing. Through a combination of these methods, a thorough and comprehensive array of tests could be completed whilst also remaining time efficient to ensure completion within the time frame.

During the project's completion, the team took part in regular continuous peer testing. This approach, whilst rather informal, ensured that all updates to the project were checked by various members of the team before being implemented. This method allowed the team of developers to review and check each other's code for errors at every stage of development whilst simultaneously offering improvements or alternative methods of implementation. This approach is viable due to the features in the team's chosen development platform: Git. It has been said that "peer reviews identify problems that can be fixed early in the life cycle"[3] of a project, meaning many major bugs can be avoided altogether or completely fixed before implementation as it is "four times more costly, on average, to identify and fix a software problem later"[4].

Unlike peer testing, White box testing was completed after each individual unit's completion to assess the performance of that specific module of code. JUnit tests were created to evaluate whether the code provides the correct outputs for predetermined inputs. This method of testing held particularly effective for the architecture of the game as there are many classes and functions which will all need to be tested independently and immediately before being implemented into the larger project. These tests were created and performed by the programmers to complete checks on their code and assess whether it performs as intended "to prevent any hidden errors later on"[5]. Another perk of these tests is that once they have been created, they can be run at any later stage of development to ensure that the functionality remains consistent throughout developmental changes. However, white box testing fails to test for missing functionality, but rather focuses on software which already exists.

This is where Black box testing is more appropriate, as it performs tests on the entire system after all the features have been implemented. This type of testing required the team to propose a set of tests and the corresponding outcomes. The main feature of Black Box testing is that "the internal structure/design/implementation of the item being tested is not known to the tester"[6]. To achieve this, it was decided during the allocation of team roles that certain members would not see the internal structure of the game to enable them to perform the black box testing successfully [7]. This form of testing is extremely effective at highlighting errors in the functionality that can easily be missed by the core programmers [8]. This accurately represents the user's interaction and thoroughly prepares the game to be released to the public.

Throughout the three previous methods, all tests were created with a close focus on requirements in order to ensure focus on the client's specification. This was specifically prevalent within the black box testing as it was rather difficult to produce concise test cases. To combat this, they were designed with the requirement specification in mind [9] and a traceability matrix [10] was produced to map the individual tests with specific requirements.

<u>Test Report</u>
As a whole, the combination of various types of tests provided a comprehensive analysis of the system and examined the functionality in all areas. Whilst the team initially encountered more errors than expected, they were quickly rectified and proved the necessity of the rigorous testing. Because of this thoroughness, the team is confident that the testing was sufficient enough to prove that the code met its requirement specifications in all applicable areas.

However, the team ran into some challenges with time constraints during the final phase of testing. It was found that certain features did not work as intended with limited time left to safely make changes to the code. Because of this, there is a possibility that fixes made to rectify the broken code at the end of development are not completed to the best of the developer’s ability. Consequently, it was decided that these aspects of the code would ideally be returned to in an attempt to optimize them during the next stages of development.

<u>White box testing</u>
Upon completion of the project, the team had a total of 7 unit tests within the game, of which 7/7 were successful.

Unit testing was not possible for all parts of our code as many of them required elements only available when the game is running (OpenGL related classes and internal files). This means that even our own Entity class, which requires a texture loaded from file, is not suitable for unit testing. Where unit testing was not possible, we have made sure to enforce stringent coding standards (req 3.3) and other techniques such as continuous integration to make sure any errors are quickly identified and resolved.

As a result, our unit tests focused on areas of our logic which were independent from implementation. We created unit tests to confirm functions used for AI logic, boat navigation and shape operations.

<u>Black box testing</u>
In total, 30 black box tests were completed by a member of the team to ensure that each specific requirement could be fulfilled. Out of these 30 initial tests, 15 tests failed. However, 10 of these failures were due to lack of implementation at this stage of development. This is due to the fact that the tests were created independently from the code and, as a result of this, it was not known which features had or had not been implemented. Despite this, it was deemed best to leave the tests in the table as to allow the clients to see which features had and had not been implemented from their initial requirements.

The remaining five failures were due to bugs in the code during the first round of black box testing. As these tests failed for features which were supposed to be functional, fixing them as soon as possible was given the highest priority. Following their rectification, the black box tests were run for a second time.

The following failures were due to lack of implementation at the time of initial testing:

Test ID	**Reason for failure**
2, 4, 5	Menu not yet implemented
23, 24	Questing system not yet implemented
25, 26	Ship upgrades not yet implemented
27	Minigame not yet implemented

28, 29	Loot system not yet implemented

And the following failed at the time of testing, but were later rectified:

15	Error in code
18, 19, 20	Collisions not coded correctly
30	Error in ship AI code

Testing Evidence

Black Box Testing: https://sepr4.github.io/web/submission/assessment2/testing/Blackbox.pdf
Traceability Matrix: https://sepr4.github.io/web/submission/assessment2/testing/Matrix.pdf

References

[1] Software Testing Standard Website [Online] Available: http://softwaretestingstandard.org/ [Accessed 30 Nov. 2018]
[2] Summary of IEEE Software Testing Standard [Online] Available: http://www.cs.otago.ac.nz/cosc345/lecs/lec22/testplan.htm [Accessed 30 Nov. 2018]
[3] A. Kolawa; D. Huizinga, *Automated Defect Prevention: Best Practices in Software Management*. Wiley-IEEE Computer Society Press, 2007, p. 261. ISBN 0-470-04212-5.
[4] D. O'Neill, National Software Quality Experiment [Online] Available: http://www.reviewtechnik.de/NationalSoftwareQualityExperiment.pdf [Accessed 30 Nov. 2018]
[5] L. Williams, *White-Box Testing* p. 60–61, 69 [Online] Available: https://students.cs.byu.edu/~cs340ta/fall2018/readings/WhiteBox.pdf [Accessed 30 Nov. 2018]
[6] Software Testing Fundamentals Website - Black box testing [Online] Available: http://softwaretestingfundamentals.com/black-box-testing/ [Accessed 30 Nov. 2018]
[7] M. G. Limaye, *Software Testing*. Tata McGraw-Hill Education, 2009, p. 216. ISBN 978-0-07-013990-9.
[8] R. Patton, *Software Testing (2nd ed.)*. Indianapolis: Sams Publishing, 2005. ISBN 978-0672327988.
[9] Element of SEPRise!, Updated Requirements [Online]. Available: https://sepr4.github.io/web/submission/assessment2/updated/Req2.pdf [Accessed 1 Jan. 2019]
[10] Element of SEPRise!, Link Traceability Matrix [Online] Accessed: https://sepr4.github.io/web/submission/assessment2/testing/Matrix.pdf [Accessed 5 Jan. 2019]