Software Testing Report

Cohort 1 - Team 2

Team Members

Camran Qadeer

Kieron Jones

Will Burden

Tabitha Oldfield

Adam Nicholson

a)

- In order to test a project made in LibGDX we used JUnit and Mockito to allow us to incorporate headless testing for the game

- To run these tests we used a “GdxTestRunner.java” class which was imported from a third party source to allow us to run the headless tests for our game
    - This code was recommended by the stakeholder and can be found at: https://github.com/TomGrill/gdx-testing

- Using this framework allowed us to write JUnit tests to test all various functionalities in our project, apart from UI elements which cannot be tested

- We also wrote manual testing documentation for UI based tests where Unit testing was not possible, as well as tests for other aspects of the game where due to time constraints, complete code coverage for modules and packages could not be achieved
    - This material can be found on our website under the “Download Assessment 2 Deliverables” Tab

- When approaching testing, we used Github Issues as well as the Code Coverage tool built into IntelliJ IDEA to track the progression of the testing process
    - For example we had different GitHub Issues set up for different packages in the project; the Game and Utils packages for example

- In these Issues we were able to create a checklist of all the classes and functionalities that needed to be tested
    - Using the Code Coverage tool we were able to see when classes and packages had been fully tested and so they could be checked off on the Github Issues

- This way we were able to keep track of how testing was progressing and what still needed to be done

- The methods stated were appropriate for our project due to the overall efficiency of having live testing information (code coverage tool) during development as well as a place to visualise and store progress (GitHub Issues)
    - We did not come across any other better method
- Having automated testing was appropriate because you would instantly be informed if any changes you make have caused an error or conflicted with something you have already done, thus avoiding these conflicts being put onto the main branch
    - You can then easily narrow down the problem and fix it
    - The tests were run on all branches, not just the main one

b)

During the testing of the project, it was important to ensure that the existing architecture was tested alongside the new additions that we had implemented. The tests aimed to ensure that all requirements that had been implemented worked as expected and that documentation existed to ensure that new developers would be able to test features of the game.

Over the course of the testing process 38 JUnit tests were written and run, where 100% of these tests passed as expected. These 38 tests were divided up as follows:

Asset Tests

- 13 Asset Tests were run, to ensure that all external assets used by the game existed in the project directory
- If a new asset needed to be added, the test for it could easily be made

Utils (Utilities) Package Tests

21 Unit tests were written and run on the Utils package to ensure that varying utilities in the game functioned as expected, which include:

- QueueFIFO Tests to ensure that the FIFO Queue data structure used in the project worked as expected.
- TileMapCell Tests to ensure the constants needed for the TileMap to work were correct.
- Utilities Tests to ensure all the utility functions used in the implementation of the project work as expected. These tests worked by producing expected results from the methods and testing these results against calculations of what the methods should do, and asserting that these results are the same.

Element	Class, %	Method, %	Line, %
Constants	100% (1/1)	33% (1/3)	3% (1/27)
QueueFIFO	100% (1/1)	20% (5/24)	14% (8/57)
TileMapCells	100% (1/1)	100% (1/1)	100% (3/3)
Utilities	100% (1/1)	100% (15/15)	100% (27/27)

Game Package Tests

4 Unit tests were written and run on the Game package, which include:

- Faction Tests to ensure that constructors for factions worked correctly
- Points Tests to ensure that the requirement of gaining points while playing the game worked correctly.

Manual Testing

- The manual testing document that has been written describes a set of tests that can be carried out in a step by step fashion to ensure that various features of the game work as expected.

- The document details 11 tests that were designed specifically to test the requirements outlined in the documentation. This means manual tests exist for:
    - Moving the Ship
    - Collection Chests
    - Colliding with Obstacles
    - Firing Cannons
    - Combat with Ships
    - Destroying Colleges
    - Changing the Difficulty of the game
    - Running out of cannonballs
    - Ending the game
    - PowerUps
    - Saving/Loading the Game

<u>Other Comments About Testing</u>

- From the automated tests that were run, 0 of them failed, however this is due to the fact that the tests written did not cover a sufficient amount of the code base.

- As described above, only 4 tests were written for the Game package, which contained 8 modules that make up most of the project.

- This amount of automated testing obviously is not sufficient and this was largely due to time constraints during development of the product.

- It was critically important that testing infrastructure for the game requirements existed as part of the development, even if it was not possible to produce automated tests.

- This led to the decision to compile a detailed manual testing document to ensure that the functionality and requirements of the project were adequately tested.

- While manual testing proves to be tedious, and can also be unreliable due to human error, it allows for aspects of the project to be tested that cannot be tested by unit tests, for example the UI module.

- Overall our testing infrastructure presents users and any future developers of the project methods to assert that the requirements of the game are implemented correctly and work as expected.

- In order to ensure correctness in our tests, each test method was laid out clearly with the appropriate JUnit libraries imported
    - Within these tests, we included brief comments about what the expected outcome was, and what the test was actually doing

c)
Link to our manual testing document:
https://eng1-c1t2.github.io/York-Pirates-2/pdfs/new/Manual%20Testing.pdf

The folder on our repository that comprises all JUnit tests we have written:
https://github.com/ENG1-C1T2/York-Pirates-2/tree/main/tests

Statistics of the tests we made:
https://eng1-c1t2.github.io/York-Pirates-2/tests.html

- This was automatically updated, as mentioned in the CI document, every time changes were made to a branch.
- Of all the tests run, it can be seen that 100% of them ran successfully.