2.1 Unit and Integration Test

Unit Tests

- Score system
    - Count total score from stepping over spawned rewards and punishments and keeps track of when the player has all required rewards to win or too few points to continue playing.
- Character and enemy methods
    - Spawns the character and enemy onto the specified position from the world loader.
    - The character class also contains a method for movement which has test files as well.
- System that loads a string from a text file
    - Reads numbers in a text file into a string to be used to render the world.

Integration Tests

- Enemy movement system
    - Enemy moves towards the player by considering the player’s path as well as a breadth-first search.
    - The key interaction is between the enemy and the spawned player. The enemy is expected to move based on the player’s position and path in all four directions.
- System that converts a string into the information about the world map including character and enemy starting positions
    - The crucial interaction is between the text file to string converter and string to world map converter.
- System for generating and rendering positions and times for bonus rewards to spawn
    - Checking if rendering the world also correctly renders where the bonus rewards are meant to spawn and despawn.

2.1.2 Test Quality and Coverage

Measures to Ensure Code Quality

We have been sure to clearly and simply lay out our tests in a very similar fashion to how our implemented code such that all tests can be easy to find. All of our tests also follow a similar coding style and naming convention so the code in them can be easily read with minimal need for comments. All tests have been reviewed and edited by our team member Matthew to ensure that all codes follow the same level of quality and style. However, it was notably difficult to implement some of these tests due to how tightly interconnected the large parts of the code were, like the enemy movement and game rendering.

Line and Branch Coverage

To note, only lines that assist in implementing methods that are being tested are considered. Lines declaring methods are not counted as lines covered or total lines in the system.

- Character entity test
    - Line coverage $= \frac{9}{11} \times 100\% = 81.8\%$
    - No branch coverage as there are no decisions in the code
- Main character test
    - Line coverage $= \frac{9}{1} \times 100\% = 100\%$
        - Extends characterEntity and only implements one method in one line
    - No branch coverage as there are no decisions in the code
- Character move test
    - Line coverage $= \frac{4}{22} \times 100\% = 18.2\%$
    - No branch coverage as there are no decisions in the code
- Bonus spawn test
    - Line coverage $= \frac{14}{21} \times 100\% = 66.7\%$
    - No branch coverage as there are no decisions in the code
- Score test
    - Line coverage $= \frac{20}{19} \times 100\% = 100\%$
    - No branch coverage as there are no decisions in the code
- World loader test
    - Line coverage $= \frac{6}{10} \times 100\% = 60\%$
    - Branch coverage $= \frac{1}{2} \times 100\% = 50\%$

- World test
  - Line coverage $= \frac{10}{27} \times 100\% = 37.0\%$
  - No branch coverage as there are no decisions in the code
- Move enemy test
  - Line coverage $= \frac{25}{85} \times 100\% = 29.4\%$
  - Branch coverage $= \frac{4}{20} \times 100\% = 20\%$

On average, the line coverage of all the tests is 61.6%. While in class we have discussed that line coverage is not the most ideal way of calculating coverage, it is interesting to note that our line coverage falls within typical blackbox testing coverage. In terms of line coverage, there is a large spread of different values as some methods were easily implemented with only one or two lines compared to others developed with up to 50 lines. Despite these shortcomings with line coverage, our testing appears to be sufficient as blackbox testing typically is.

The branch coverage has an even larger discrepancy between the only two tests that test for branches. The move enemy test has many if-statements with some being embedded. As such, we ended up only testing the conditions where the movement was true. This was also done as the true conditions were much more important in the context of a game. In hindsight, testing the false and combination of true and false conditions could be beneficial to checking if the method's logic is fully correct. As well, it would increase the line coverage since this method's implementation had the highest line count. The world loader class includes a Java exception that checks errors and only the case where no errors are present is tested. Another test to note is the bonus spawn test since while it does not test for branches, it does test the different conditions of whether the bonus reward has appeared or has disappeared.

Features or Code not Covered

Methods in the JavaFX file do not calculate or compute anything so they cannot be meaningfully tested. These methods are responsible for displaying the menus and UI consisting of various labels, images, and buttons.

2.1.3 Findings

What Have You Learned

Testing some things can be very hard because of how closely coded some large aspects of the program are like the enemy's movement or the game rendering. Because of how dependent these objects are on each other, it is difficult to run only a small part of it without requiring the entire rest of the program to run. And when that happens, everything is being tested rather than the initial small element.

Changes to Production Code

There were many little changes to our production code but many of them were small and simple changes that didn't change how the game worked overall. This includes things such as changing or adding comments, changing variables names, and other minimal changes. There was one notable change which was moving the enemy movement class to its own file. This was done so that it can be individually tested.

Fixed Bugs/Improved Quality

By going through the code again with the purpose of testing rather than making sure it works, there were many places where poor programming style and coding inefficiencies are seen. Unlike while creating the game, we are more inclined to fix these problems since there is no need to worry about getting the feature working in the first place. This includes things such as separating some classes into smaller, separate files or taking out hardcoded magic numbers. One example of this was that the enemy spawn position was hardcoded right into a method rather than storing that information in our world text file. To clarify, this text file holds all the information about the map to prevent splintering information across multiple classes and methods. Making this change to store it in the world text file allows for easier changes of the map without worrying about why the enemy is in the wrong position.

To expand on separating our code into smaller classes, it is not only to make the code much easier for us to read but also allows for individual testing to run on smaller separated code rather than a large block. For example, removing the enemy movement function and implementing it into its own class so that it could be tested with it's own special world and spawn locations. This is to not be concerned with making it work in conjunction with the main class that renders our predetermined map.