Phase 3 Report

JUnit Tests:

Here are all the main features that we tested with the help of the JUnit test:

1. Testing the position class, which gives the functionality to store data in terms of coordinates such as storing the position (30, 60) as (1,2). This is very beneficial because the pathfinding algorithm stores the whole path as coordinates. Therefore, we test it to see if it creates the objects without any issues while having the functionality to convert positions in integers. All the stated functionalities are crucial for the enemy's movement. As a result, we test its feature for making sure that its functionalities work properly.

2. Tested the pathfinding class, which is responsible for finding the shortest path between the enemy and the character. Therefore, the function finds the path by taking the position x, y of the enemy and target (Main Character) to find the full path with the use of the BFS (Breadth-first search) algorithm. As a result, for testing this feature we have used Unit testing by generating different positions for enemy and target to cover the 100 % of the branches in the algorithm.

3. The collision functionality between moving entities such as the main character and the walls/barriers has been tested. Testing this feature, only requires giving it the x and y position, which is the location that the main character wants to go to. After giving the required parameters, it will check if the given moving entity's position is going to hit the wall/barrier or not. Therefore, we have tested the different scenarios by generating different positions such as giving the mock position when the user wants to go out of the map. Consequently, we expect the result to be false, which does not let the user go out of the map. Hence, we have tested this functionality with assertion.

4. The collision functionality between a static entity (regular rewards/punishments) and the moving entity has been tested. We have implemented a Junit class, which creates mock positions for each object while knowing that based on the indicated positions whether the collisions happen or not. As a result, we have given the data to the method and got the results that we have compared to the expected results to make sure the collision functionality works as expected.

Integration Tests:

1. The interactions among entities, map, and collision classes have been tested by checking if the score of the player gets updated when it collides with static entities (Rewards). For testing this functionality several classes interact with each other to update the score. As a result, we have given the player (Main character) mock positions based on our knowledge about the map by knowing the rewards positions. To check whether the collision happens between the two objects if the score gets updated or not using assertions.

2. When the player wins, several classes are interacting with one another such as the game state and the game map, which in each tick check if the player is winning. Moreover, objects such as the Main character and static entities are interacting with the Game map since they have been placed on it. Therefore, we have tested different scenarios such as checking whether the player wins when all regular rewards are collected while the player is at the finished sign.

3. When the player loses classes such as Main character, enemy/punishment, Game map, game state, and collision are interacting with one another. Therefore, to check the interactions between these classes we have set mock positions for player and enemy to interpret in different scenarios. For example, testing if the player loses when the enemy and the player collide with each other.

4. The interaction between the enemy and the pathfinding class based on the enemy's movement has been tested, since its movement is dependent on the given path. Therefore, these two classes use and interact with each other. So, we have given the enemy and target (Main character) different positions on the map. Based on the expected movement direction of the enemy, we have tested to ensure that both enemy's movement and pathfinding interactions are working properly.

Discussing the measures, we take for improving the quality of tests:

Here are some measures we toked throughout this phase for improving the quality of the tests:

1. In each class we have documented/commented on the step-by-step descriptions of the logic behind the tests and the specific task, which we want to verify with the help of JUnit/Integration testing. Therefore, writing the complete instructions should make the job of maintaining the test class by either adding more test cases or extending it easier in the future.

2. Improved the clarity of the code by using appropriate name conventions for each function in test classes has been used to improve the understandability of the code.

3. We have written our test classes in a way that it is easy to add additional test cases without the need to do any duplication, which can save us time in future. For example, we have used parameterized test that gives different test cases as an array list for testing features in our code that takes as parameters. Therefore, it makes testing different branches and scenarios easier since you only need to increase the array size if you want to add more test cases. As a result,

adding or changing any of the stated test cases in classes is fast and efficient without requiring any code duplication.

4. Designing sufficient test cases for making sure that all the branches/decisions in a program will be covered. Therefore, for being able to cover as many branches as possible we have had to choose the important test cases. As a result, covering near 100 % of the branches can give us more confidence in the correctness of the implementation.

Which features or code segments have not been covered in tests?

Although most of the features have been covered in tests, here are some features that we could not test for different reasons:

- The movement of the main character is dependent on the user input from the keyboard. Hence, it was difficult for us to test it. With that said, we have tested all the possible collisions by generating mock positions for the moving character to make sure that when the character hits the wall/Barriers/rewards, the collision gets detected.
- The graphics of the game such as rendering images into the screen. Because it uses the graphics object, which is private and difficult to have access to test. Moreover, the rendering for each object is done by void functions. So, the rendering function does not return any output. Therefore, we did not cover it in our test classes.
- The mouse manager, which is responsible for inputs from the user's mouse for interacting with menu features. Due to its dependency on inputs from the mouse, we find it difficult to test.
- Key Manager package that takes the key inputs from the user keyboard and sets the up/down/left/right to true or false accordingly. Therefore, due to its dependency on user's input, we did not test its functionality.
- Game class, which contains the game loop has not been tested since it's a void function and does not take any inputs or gives us an output. Therefore, it was untestable.
- All UI features such as buttons and states have not been tested since they require more complex testing such as systematic tests.

What have we learned from writing and running tests?

Here are some important points we have learned during this phase about tests:

- While writing tests can be easier and faster than the implemented code, the quality of tests plays an important role in detecting bugs in the source code, which shows the importance of running and understanding the writing process of tests.

- Functions with complicated logic while having many dependencies can be very difficult and sometimes impossible to test.
- Writing tests in order to cover the 100 % branches of complicated algorithms can be challenging since we can have a lot of conditions and many different paths. Therefore, by choosing the important test cases we must ensure that all branches of a code get covered for gaining confidence in the correctness of the code.
- Tests can be time-consuming since by detecting a bug or an error you have to make modifications to your production code.

Changes to the production code during the testing phase:

During phase 3 we changed the production code and improved it in terms of both efficiency and design. Here are some of the modifications we made to the production code:

1. Improved the collision functionality between moving entities and obstacles such as barriers/walls. The function has been modified in the collision manager class.

2. Changed the entire pathfinding logic of the enemy since we wanted to improve its design and efficiency while making it easier to test.

3. Have added a menu that has three buttons including exit, how to play, and play the game. The users simply by clicking their mouse can easily interact with these features. For instance, by clicking the play game button, the game starts, and the user can play the game or by clicking the exit the user will exit the game.

4. Added how to play state into the state's package, which gives instructions to the user on how to play the game while it has a menu button that users can easily access by clicking on it, which returns to the menu.

Were we able to reveal and fix any bugs and/or improve the quality of code in general?

Yes, the following points are the modifications and improvements we made to the production code:

- We found several bugs in the enemy's movement and its path-finding logic. Therefore, we decided to completely change its logic in order to improve its code quality and correctness.
- Improved the efficiency and quality of the collision's functionality between moving entities (Main character) and walls/barriers by placing the method in the collision manager package and modified its logic.

- Changed the design of the game map from singleton since we found it to be inefficient and difficult to test.
- Improved the quality of the game class by deleting duplicated and redundant code for simplicity and having a better design.