Unit Tests

Character:

For the character class, our group focuses on the character's response depending on the block that it moves to. This leads to six different scenarios based on whether that block contains:

- A reward: the character needs to check whether the reward is a basic or special and add points accordingly. If the reward is special, it will not count to the progression of the game.
- An enemy: the player loses.
- A trap: the player will lose 5 points. If this results in a negative total point, the player will lose.
- An exit: if the player collected no reward, only special rewards, or not all basic rewards, the game proceeds. The player only wins if they collect all rewards before stepping on the exit.
- A wall: the player will not move and their coordinates are not changed.
- Nothing: the player's coordinate changes based on the key pressed. However, the points, the game state and the progression toward winning does not change.

Enemy:

The enemy is tested based on the path it chooses to reach the character based on the wall setup. To do this, we compare the number of steps needed to reach the character, calculated by the algorithm to the smallest number of steps needed. We tested this based on five different scenarios:

- There is nothing between the enemy and the character and both are on the same row or column.
- The enemy and character are on different rows and columns.
- There are traps and rewards between the enemy and the character.
- There are walls between the enemy and character
- There are walls between the enemy and character, arranged in a way that forces the enemy to move away from the main character.

Leaderboard:

For the leaderboard, we tested it based on its ability to:

- Check whether the latest run is in the leaderboard: to check whether the latest run should be in the leaderboard, there are four different scenarios:
    - The latest run is not in the leaderboard
    - The latest run is in the leaderboard, but it is in the last place, which means there is no shifting between the existing highscores
    - The latest run is in the leaderboard and it is not in the last place, which means there is shifting between the existing highscores. However, the list is not full, which means no entries get pushed out of the leaderboard.
    - The latest run causes a shifting in the current leaderboard and pushes out an existing entry.

- Arrange the highscores in the right order: to check that all entries are in the right order, we insert 100 different entries with random speed. Each time, we check whether every entry in the array is in the right sequence.
- Write the highscores to a text file.
- Read the highscores from a text file.

Reward:

To test the Reward class, we created a map that is half-covered by walls, then we insert 10 randomly-placed rewards and check whether they are spawned on top of a wall or each other. Since there is an element of randomness with the algorithm, we do this 100 times to make sure that there is no error.

Integration Tests

Inventory, Power Ups, and Interface:

One of our integration tests include MechanicsTest.java. This test aims to integrate the different mechanics of our game, and test the communication between the modules. The tests both the inventory item and powerup system as they are separate classes, and communication between in the character class was important to get these mechanics working in our game. Since the data was hard-coded into an array during gameplay, it was unnecessary to repetitively test with different types, for example:

- Testing if different power ups are added to the array since all of them utilize the same interface (inventory.addItem(), etc)

The result from these tests did not discover bugs, however, we were able to analyze the communication between the three classes to design a better system that would be able to utilize these mechanics in the character class. For example, instead of creating multiple instances of InventoryManager or PowerUps in both the GameScreen and Character classes, we can reference it only in Character and then use a separate rendering class to create the labels and add them onto the game screen.

Interface and Logic:

It was also ideal to include the user interface testing into this main integration test, however, it was difficult to test using maven as we would have to manually create and update the JLabels that weren't actually being rendered since the tests did not start our game. Instead, we chose to apply a different method of testing that included playing the game with specific test cases in mind to test if the user interface was updating correctly. Some of these manual tests included:

- Manually placing multiple power ups to test if the inventory system would not allow power up pickup when the inventory was full
- Checking if the inventory render updated the correct slot with the correct power up
- If the label correctly updated on power up consumption, and if it was mechanically updated (did not exist in the inventory array)

An additional interface test that would utilize multiple modules of our game include such as game state and maps include:

- Repeated maps: making sure the maps rerender on map switch and if the game state updates correctly ones the player loses or wins the current playthrough
- Logic connection with UI: does the timer, map, and inventory system correct reset and update upon game state switch?

These additional tests were also conducted using manual methods due to the limitations of the automatic testing in maven. Some minor visual bugs were found and correctly updated in the class files.

Findings

To make sure that we cover all of the branches for every class, our group drew a control flow graph for each method that the class has. Since we gave every object a number, it is very easy to keep track of all the possibilities when an object interacts with another. However, we decided to not test some minor classes such as Point, HighScore and Settings since these are used to support the major classes that we already test. Additionally, the tests for those major classes already include the methods of these minor ones. We also decided to not test simple methods or ones that only have one outcome. For example,we wouldn't test a method that is only used to set the image of a class. This is so that we can focus more on the other methods that require more time.

Testing allows us to find errors that we couldn't find in phase 2. For example, we found a bug which makes our program treat the last basic reward in the reward list as the bonus reward if the player collects the bonus reward in the run. We couldn't find this bug during phase 2 since we rarely collect the bonus reward when we manually tested the game. However, with automated testing, we could find this bug since tests covered every possible case of the methods. Additionally, testing allows us to modularize the code so that we can test the smaller components of an algorithm. As a result, debugging is much easier for our group in phase 3 since we can know exactly which parts of the algorithm are not working as intended. Furthermore, due to testing, we were able to make most of the hardcoded part from phase 2 more dynamic since we needed to test with different inputs.