Group #13 - Phase 3 Report

Xuyun Ding, Denzel Tjokroardi, Kyle Tsia, Yihao Wang

Unit Tests :

RewardsTests

Within this test case, we test the three states that the Reward can be in, *Mandatory*, *Bonus*, and *Dead*. First the constructors are tested, *Dead* rewards are not constructed in our game, so there are only two tests, one for *Mandatory* rewards and *Bonus* rewards. Then we test the basic getters of the class, we check that each getter retrieves the expected value. Finally, the last two tests are the state setters, one for *Bonus* rewards and one for *Dead* rewards. Both of the final tests for when the state of the reward changes from *Mandatory* to either of the other two, and once again we check that the values are the same as expected.

Cell Test

Within this test case, we test the setter and getter for Cell class. These two features are used to set the type of the cell and get the type of the cell. We only use one test case to test two features because they are related so it is sufficient to cover all situations.

Tile Test

For this part of testing, we decide to use two test cases to examine the initialization and the construction of the tile. One of the test cases show that the tile can be created successfully and the other proves that the created tile is of the same size as a 32*32-pixel, which is exactly what we have planned.

Board Test

To make sure that the *Board* class works correctly, we apply three cases to test its basic functionalities. First and foremost, the first test case is used to prove that the map file can be successfully read in from the resource folder, which is the foundation of the maze construction and collision detection of the game. After this, we use the other two test cases to ensure that the read-in map file is of the right size in order to let it show up correctly in the GUI later.

SpriteSheet Test

Since the *SpriteSheet* class mainly deals with the icon pictures of different entities in the game, we create four test cases here to ensure that everything works as intended. The first case is to make sure that the entity picture can be successfully loaded from the resource folder for later use. The other three test cases are used in order to see whether the *getPicture* method will retrieve the

correct icon pictures for the hero, enemies and punishments respectively from the right locations with the right sizes.

Game Test

In the game test, we have both unit tests and integration tests. Here we discuss the unit test for our game. We test the constructor, start, stop and tick. In the start and stop test, we check the state of our game if it is running. In the tick test, we check the time of our game if it is adding 1. In the constructor test, we check the state of the game if it is in the START mode(we have four modes which are START,GAME,END and GUIDE). After these tests, we corrected the game class to make it work and execute in a healthy way.

Player Test

In the *PlayerUnitTest* class, we test whether the constructor initialized the player cell in the correct coordinates, started the score with a 0, and initialized the round count with a 1. We also tested the *scoreChange* method where an assertion was made to make sure that the PlayerScore variable stored in the Player object was correctly updated.

Enemy Test

In the *EnemyTestUnit* class, we tested both the *Enemy* class as well as the *MovingEnemy* class the constructor's initialized values of the coordinates as well as asserting that *valPunish* is a random value that is less than 3 and that *visited* is false. We then also tested to make sure that both *enemy* and *movingEnemy* updates correctly when visited is set to true.

Integration Tests:

Game initialization with player,enemy and rewards

We already have unit tests for Game and we also need integration tests for Game. But we can't test all the classes in the game by unit and integration test, especially using integration tests. Because if we want to test render() or run() in game, we must make the game in GAME mode which means the game is playing by the user. So we choose to test on initialization. For the initialization, it will relate to the constructor of player, enemy, rewards and several types of List, we have 5 test cases. The first test is to check if it constructs the enemy List. The second test is to check if it constructed the movEnemy List. The third test is to check if it set up the mandatory reward List and make sure they have the correct parameter. The fourth test is to verify the size of the mandatory reward List. And our last test in gameInitialization is to verify the size of bonus

rewards.We also can't test the key listener part because it needs the input from the keyboard.

Maze construction with Board/Tile

In order to make the maze construction successful at the beginning of the game initialization, we need to make sure that the *Board* class and the *Tile* class work together well. The first test is to show that for a single black grid (here we choose the initial one) on the pre-made map picture, the *Board* class will be able to identify it with color detection and then make the corresponding cell on the game board to be a tile. The next test case examines the total count of tiles on the game board, which are the walls/barriers of the game and should match with the number of black grids on the map picture in the resource folder.

Collision detection between walls and the hero

Next fundamental functionality of our game is the collision detection between the walls and the player which requires both the *Board* class and the *Player* class to work properly together. We implement the first test case to ensure that the spawn point of our hero will not be in any cell containing walls or barriers. Then we create the other two test cases to choose two points on the game board to see that a move of the player to a cell with a wall or barrier can be detected as a collision, and later on a false boolean value will be given to the *tick* method in the *Player* class to ignore this illegal move.

Collision between Player, Enemy and Rewards

It is imperative to also test the interaction between the player, enemy and rewards objects as this is the winning and losing condition of the game. Therefore, we implemented 2 integration tests, one of them are for the cases if an enemy collided with a player and tested the score decrease functionality as well as testing the case when the movingEnemy is encountered, where the game will end immediately. The other test is done to make sure that when a reward or a bonus reward is encountered by the player, the player score will update as expected.

Findings & Adjustments

Within the aspects of maze construction, entity icon loading and the collision detection between the player and the walls/barriers in our game, we have not found any major or fatal bugs or errors so far. In order to enhance the stability of our game and decrease the complexity of our code implementation, we have used all separate pre-created pictures for the maze and different entity icons. Therefore, the core mechanics become the file read-in, color detection for tile/board as well as the collision detection between the player and the walls. As

all of them are pretty straightforward without too much logic complexity and do not intertwine with each other a lot, we decide to leave the code for these aspects as it is from phase 2.

We find a problem in our phase2 java code in game class by game tests. When we test game.stop(), we find it will always fail to test because the old game.stop() method has a logic error in if(isRunning) which should be if(!isRunning).After we finish all the tests for game class, we learn how to write tests when the test will relate to other classes and know how to design the test case to test one class in different dimensions.