CMPT276 Phase3 - Report

What we have learned from writing and running test:

What we learned from the test is how to contribute to the question of the function unit in the test. In the functions, the problems seem not so easy as the question example post in the course. In this test I was responsible for the test of the Enemy class, and one of the most important activities of that is the testing of the AutoFindWay method. In order to test the path find function, I have to know the test case, where the start should go and where the end should be and what are the length of path between them. It needs the design of the map and checks the length through a definite correct method in order to ensure the result of the auto path find function is right. Even if it is hard to do, we still design the test case to let the function find one of the longest paths from the start of the map to the end of the map.

The other thing we learned is choosing the class which needs the test. In this application, most of the work in the GUI is based on the APIs built in the Java, they are not testable with the simple test of program, but need more technique in UI testing. Out of that, some classes work in the other class, which can not be tested directly as well. We use the design of the application to clear the dependency to get which class can test directly and which class should be tested with the other classes indirectly. With the design of the class, we also get which function doesn't touch the GUI APIs and can be tested with JUnit. In this case we learned how to drag the important information in the test from the design we made initially.

Test case/class mapping:

Test class: Character--------->feature: get the bonus from the map, move property, follow the command from the keyboard adapter function.

Test case: Let the character move from the start point to one direction strictly and get the bonus on the way. The bonus has already been set in the way of character-------> feature: Character can move forward and collect the bonus correctly.

Test class:actObject---------->feature: record the position of the active object(Character and enemy), and move following the command sent by keyAdapter function.(actObject is an abstract class so that cannot test directly, it should test with the instance of Character.)

Test case: Move the character in the up towards------>feature: character can move in up toward.
Test case: Move the character in the down towards------>feature: character can move in down toward.
Test case: Move the character in the left towards------>feature: character can move in left toward.
Test case: Move the character in the right towards------>feature: character can move in right toward.

Test class:Enemy---------->feature: find the path from the input start point and end point, move itself automatically follow the path calculated by the path_find function

Test case: Let the enemy find the path from the start of the map to the end of the map, and calculate the length of the path.--------->feature: Enemy can treat the wall as the constraints of path and enemy

can calculate the right distance from the start to the end of the map through the path it finds.

Test case: Set the character in the sense area of the enemy, and let the enemy move automatically to catch the character---------->feature: Enemy can follow the optimized way it calculated correctly and get close to the character and catch it.

Test class: PrizeFactory---------->feature: create the prize properly with the assigned prize type. And create the array which stores the information of the prizes with different types. (The prize class constructor is not visible to the test class so that we have to use the factory to get the prize product)

Test case: Let the PrizeFactory create the reward-type prizes and store them in the array properly--------->feature: PrizeFactory can create reward prizes correctly and put it to the storage array correctly.

Test case: Let the PrizeFactory create the bonus-type prizes and store them in the array properly--------->feature: PrizeFactory can create bonus prizes correctly and put it to the storage array correctly.

Test case: PrizeFactory can create prizes with the coordinate of map input------->feature: PrizeFactory can create prizes at the correct place in the map.

Test case: PrizeFactory can create prizes - type reward with the value matched with the input------->feature: PrizeFactor can create prizes with the correct value that provided

Test case: PrizeFactory can create prizes - type bonus with the value matched with the input------->feature: PrizeFactor can create prizes with the correct value that provided

Test case: PrizeFactory can create prizes with the status active (equal to true)------->feature: PrizeFactor can create prizes with the status active (equal to true)

The result of the test:

1. Character can work on eating the prizes properly, and calculating its score properly according to what it eat

2. The moving of the actObject can move itself following 4 kinds of commands properly.

3. The enemy can calculate the right path from the start of the map to the end of the map.

4. The enemy can move properly and get close to the character

5. The PrizeFactory successfully creates 2 arrays of type reward and bonus. Each prize objects has a coordinate that matches with its position pre-set on the map. Each prize objects' status is active after it's created by PrizeFactory.

What didn’t cover yet in the test:

1. The whole process of a game plays from moving the character, collecting the prizes and moving, avoiding the enemy and finally finishing the game.

Reason: The game playing process should be worked with the keyadapter and the GUI API, the technique right now we have is not enough to test these things.

2. The gaming reset function.

Reason: Before resetting the game, we have to disorganize the content classes in the map including the character, prizes and enemy. But at the very first beginning, we don’t design such a function in the game, either the game is not designed in the form of being able to run the playing script. In the other words, it is hard to operate the map without the KeyAdapter API, but the API testing is also not able to test with the technique we have.z

Were we able to find and fix any bugs during Phase 3:

At the end of Phase 2 our code was not a fully working game due to problems with the game engine. After meeting with our team we were able to come up with a working game by reworking our game logic for drawing the graphics and capturing keyboard inputs. Additionally we went over the method and class names to ensure that they followed the Java naming conventions.

In revealing and fixing the bug is not so complete in our game. Because we rebuilt the game engine of the game, which wasted a lot of time, and doesn't have enough time in constructing the complete exceptions catching structure in our game, and lack the catch and final resolvent and report of the bugs. But in the following development, the exceptions and bug catching structure would be built.

More important things:

As for the more important things, one of them is the exception catching system is one of the most important parts in the game debugging. Most of the important parts should run with the exception of catching and bug resolver to prevent the game from crashing. And the other thing is about the controlling visibility of class in Java. At first, every data in the game is public as the calling of each class data is easy. But the game is easy to crush by using the Cheating Engine to modify the program data. But after controlling the visibility of class, the security of the game is largely improved.