Phase 3 Report

(Group 11)

For our testing, we first had to create a mock-up of what features of our game we wanted to test separately. We wanted to test the code that would ruin the aesthetics and overall gameplay experience if they were missing from the game. This way we could improve the quality of our code significantly if we focus on the necessary components that make up our game. This includes the player, wall, static and dynamic enemy and the reward class. They are all part of what makes our game function properly and a great start towards what needed to be tested.

For the player class, we want to test the features including player creation, player position and speed. For the player creation, we want to ensure that a player object is created when the game starts and not end up being null and display nothing on the screen. We will use the assertNotNull to test this by creating an object and giving it an x and y position to see if an object is made or not. We also want to test player positioning. The player's position needs to be where we want it to be on the matrix. On our matrix, each index has a certain number associated with it and for player it is the number one. Every matrix index is forty by forty and so we want to see if the player's x and y position is where we want it to be on the matrix associated with one. We will use the assertEquals to test this by creating our own player object and getting its initial x and y position and compare it with our expected value of where we expect the player to be positioned on start up. Next, we want to test the player's speed. When the user clicks a certain key, the player's speed should move the player's position to another spot. We then will test that our getBounds() method is functioning properly by comparing a player's rectangle bound with our expected rectangle bound.

For the ScorePanel class, we want to test the features including the score number, the calculation of the score and score creation. For the score number, we want the objects to display the right number onto the game screen. We will use the assertEquals that will compare our expected integer value with the actual value. We will create the actual value by first creating a scorepanel object with two integer argument values: one for score and one for credit. We then will create two separate cases that will compare each argument using the class method getCredit() and getScore() respectively. We will then ensure score calculation is correct. Again, using assertEquals, we will check our expected value to be the same when we subtract or add ten to the initial value. We will perform this arithmetic by using the scorePanel classes reduceScore() method for subtraction and again calling the getScore() method to compare the

actual with the expected value. This will be the same with addition except we will use the addCredit() and addScore() methods. We then want to check that the score panel is displayed. We will do so by using the checkScore() and checkCredit() method and comparing it with our expected value of true that the panel is made and will be created.

As individual modules may work fine individually, we may be faced with faults when put together. We must perform integration test to ensure that the individual modules when tested together is what we expect to see in the game interface. We must first unit test modules individually and then slowly integrate all the modules together. From this, we will be able to see the difference between the expected behaviour to what is actually being displayed. We first need to create a test plan. Then create test cases. After that is done, we will integrate the components to execute our test cases. We will continue this test cycle until the components have been successfully integrated. Following this, for our integration testing, we want to examine how well our individual unit test work together and compare expected behaviour with actual behaviour. We want to ensure that end-users will not encounter any problem when playing our game. We will first examine user interaction with the system and how their input creates change to the game. As well how their input can affect the game objects and how they interact with each other.

We want to ensure that every entity on the board is where it belongs in our matrix and is performing what it needs to perform. A key interaction that needs to be tested is player movement. This is significant as the end-users only mechanical instructions is to use buttons to move the player. Our group will test the button interaction and how it changes the players movement. Movement is associated with speed and so we will test how players button press will change the speed negative or positive. Depending on the speed, we will be able to move the player character either up, down, left or right as well speed being zero if no action is performed by the end-user. We want to ensure that what the player's button key pressed is going towards the direction they are expecting as this can be a major flaw in our system if not fixed.

Adding to this, we want to ensure that the player's character is not collided with walls. Every object is surrounded by a rectangle using the rectangle function. We want to test that the user's player character remains in the same position if the next matrix index is occupied by a wall object. We want to ensure that characters including dynamic enemies are not occupying the wall index. Another thing we must check is that all objects are part of the matrix and is occupying its id numbered index spot. We must ensure that every object is instantiated in its correct position on the map to prevent overlaps and collisions on start-up.

In unit testing, code coverage is often used as an indicator to measure the quality of the test. As such, we improved the code coverage to detect the logic and operation of the code, and further modified our code as a result. We tested it in three different ways:

1. Statement Coverage

At the beginning, we used statement coverage to measure whether each executable statement in the tested code had been executed. In the IDE we used, there is a very convenient function. After running the test file, we can see how much content has been covered. In this way, we can clearly know whether there is code that has not been detected. But such detection is very weak, it can not detect all the bugs. In order to find a better standard, we also used the following detection methods.

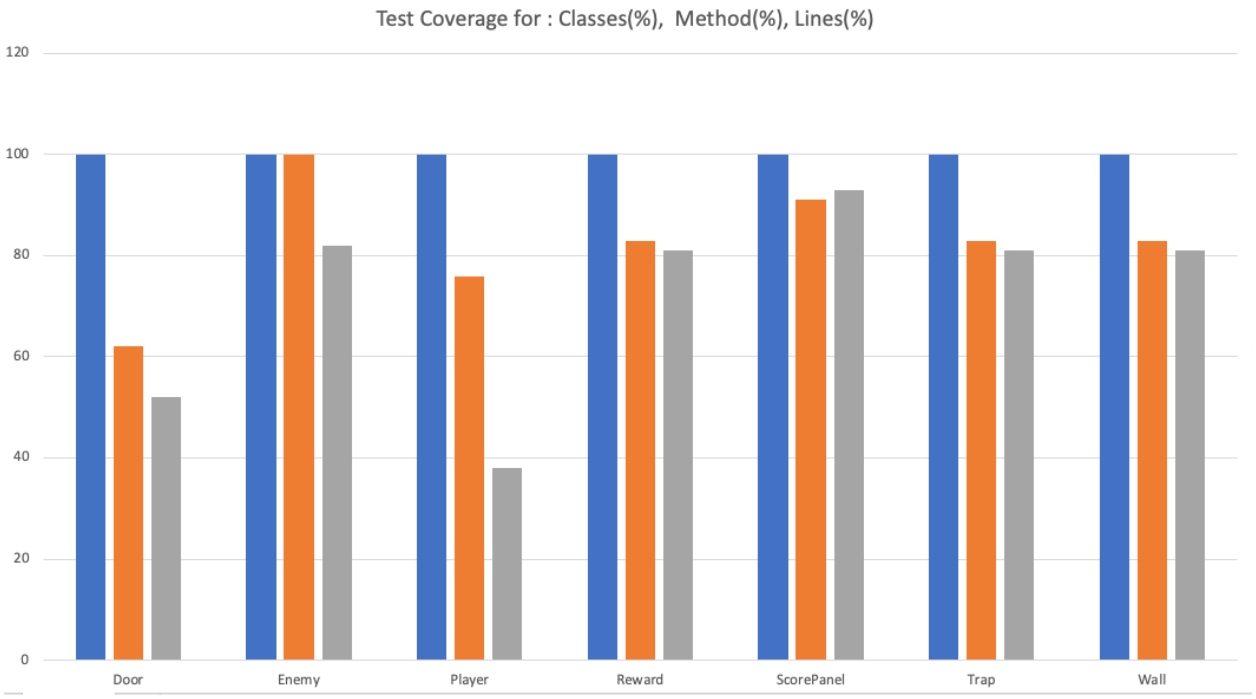
2. Branch Coverage and Condition Coverage

Condition coverage measures whether the true and false results of each subexpression in the determination are tested. Branch coverage measures whether each determined branch in the program has been tested. When designing a decision coverage case, we only need to consider two cases: true and false. When designing condition coverage cases, we need to consider the result of each condition expression in the decision.

3. Path Coverage

We also used path coverage for testing. Path coverage measures whether every branch of the function is executed. We consider that many codes have different branches, so we need to execute all possible branches once. When there are multiple branches nested, we will arrange and combine multiple branches to test one by one.

Here is a picture of our test file coverage:



In our testing process, we learned that code coverage can tell you where the product code is still, which functional modules are not tested, or very carefully tested. But it can't tell the tester whether the test effect is good enough, and the code coverage rate of 100% can't show that the user's requirements have been considered and tested. Therefore, we need to use code coverage reasonably. In the process of testing, we can specify a lower limit of test coverage. For example, the coverage of all packages and classes must be more than 80%. But at the same time, we should not only pursue high coverage, but also pay attention to the quality of test cases. If there are errors in the test cases themselves, even if the test coverage is very high, it is meaningless. All the premise is to ensure the effectiveness and quality of unit test.

While running the test for our code, we came across many errors in our work. We initially always had our player character hit a wall earlier than expected. We were not sure why this was the case. However, after testing we noticed that some of our wall object x and y values were slightly off and displaying a wall in different positions than we had expected. Although this in some cases be a minor test, it showcased that hardcoding is bad practice due to human error when assigning values to objects manually. We then improved the quality of our code by restructuring it and creating a matrix. Each index on the matrix had an associated id number that was used to instantiate different objects with. This way we can keep track of where an object is positioned and dynamically change it if we choose to do so. Through this process, we discovered that the player’s image was not the correct size as each matrix index was forty by forty and so affected our map size.