Unit Tests

- **Feature 1: Wall detection**
  - Description: The first features to be unit tested were all the boolean methods in the WallDetection class. We decided to test this feature because without wall detection, we would not be able to create a maze map and therefore we need to ensure that the wall detection is working perfectly.
  - Test case/class: We created four test methods, one for each direction that Player moves in.

- **Feature 2: Winning/Losing conditions**
  - Description: The second feature is the detectWin() method. This is an essential feature of the game because this method controls whether Player is still in a game, wins a game, or loses a game based on different conditions.
  - Test case/class: To test this feature, a class called TestDetectWin is created with six methods that each contains different test cases. The first two methods are testStillInGame(), which test two scenarios where Player is still in a game. The next three methods testLose(), which test three scenarios where Player loses a game. The last method tests whether Player wins a game. It only has one test case, which is when Player reaches the exit cell and has collected all the regular rewards.

- **Feature 3: Reward and punishment random generator**
  - Description: The third feature is the random generator that randomly places the rewards and punishments on the map.
  - Test case/class: To test this feature, a class called TestItems is created. A test method is used to test whether the rewardGenerator() method generates the correct number of regular rewards every game, same as for punishments.

- **Feature 4: Score methods**
  - Description: The fourth feature is the updateScore() methods that either increase or decrease Player's score.
  - Test case/class: To test this feature, a class called TestScore is created. Two test cases are used for testing if the score for collecting a regular reward or bonus is increased at the correct amount and one test case is for testing if the score for running into a punishment is decreased at the correct amount.

- **Feature 5: Image files**
  - Description: The fifth feature is all the images used in the game.
  - Test case/class: To test this feature, a class called TestImages is created. One test case is created for each image used in the game.mv

Integration Tests

- **Interaction 1: Collision between rewards, punishments, and Player**
  - Description: This feature involves interactions between multiple classes, including Item, Reward, Punishment, Bonus, and Player. To ensure that collision detection between Player and the items work properly, the items first need to be placed on the map correctly with the specified position. Then, for Player to move to the correct position, Player's move() method has to work and Player has to return its x and y positions correctly. Lastly, using the x-coordinates and y-coordinates returned by the items and Player, we can test if the items' boolean collision detection methods are working properly.
  - Test case/class: To test this interaction, a class called ITCollisionDetection is created. The class contains three test cases, one for each type of item (regular reward, punishment, and bonus). For example, for the TestRewardCollision test case, we added three regular rewards and stored them in an arrayList because this is what we are doing in the production code. Then, we get the position for each regular reward and compare it to Player's current position by passing their coordinates to the Item class' boolean collision detection method. If the coordinates match, meaning that Player and an item are on the same cell, the boolean should return true and pass the assertTrue() test. Otherwise, it would return false and pass the assertFalse() test. This integration test covers 100% of the Item class, including all four branches for the boolean collision detection.

```
@Test
void TestRewardCollision() {
    item.add(new Reward(5, 5));   <- rewards are added correctly
    item.add(new Reward(1, 5));
    player.move(5, 4);   <- Player is moving correctly

    hit = item.get(0).detectCollision(player.getPlayerX(), player.getPlayerY(),
            item.get(0).getPosX(), item.get(0).getPosX());   <- Collision detection working properly
    assertTrue(hit);

    hit = item.get(1).detectCollision(player.getPlayerX(), player.getPlayerY(),
            item.get(1).getPosX(), item.get(1).getPosX());
    assertFalse(hit);
}
```

Figure showing integration test for testing collision detection between Player and the items.

- **Interaction 2: Moving Enemy tracker**
  - Description: This feature involves interactions between the player, enemy, and cell classes. We want to show that the enemy is able to determine where it should move correctly based on where the player is. In order to test this we utilized the player's move method to place a player on the board. We then run the beacon method to modify the weighted map which the enemy uses to track the player. Lastly, we move the enemy using the update method.
  - Test case/class: To test this interaction, a class called ITEnemyTracker is created. The class contains one large test case which makes sure that the Enemy is moving in the expected X and Y coordinates for what is effectively 6 Ticks. We know which path the enemy must take because the player remains stationary during the tests. Therefore we have a series assertions that checks that the x and y coordinates of the enemy match what we would expect after each update.

- **Interaction 3: Main Frame**

  - Description: This feature involves interactions between the Main and Frame classes. We want to make sure that the Frame Classes and Main Classes are producing the expected objects.
  - Description: In order to test this interaction we created a class called ITMainFrame. In this class we have 2 tests, the first test creates a Frame and we assert that it is not Null. The second test uses the main method for the Frame class.

Test Quality

After measuring the coverage with Eclipse it was found that our tests had 74.6% coverage. Diving into the metrics we see that the bulk of the missed instructions exist in the Cell class, making about 41%(459/1111) of all missed instructions(See Figure 1). We dove deeper into the code and found that the bulk of all missing instructions were due to not testing random generation of bonus rewards for the map and not testing drawing methods because were unable to test Java Swing components. In addition there were some private variables that we were unable to test.

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
project	74.6 %	3,257	1,111	4,368
src/main/java	63.2 %	1,904	1,111	3,015
board	62.8 %	1,310	776	2,086
Cell.java	55.9 %	581	459	1,040
Board.java	19.9 %	28	113	141
Player.java	77.2 %	277	82	359
Enemy.java	86.4 %	342	54	396
Bonus.java	51.2 %	21	20	41
Punishment.java	51.2 %	21	20	41
Reward.java	51.2 %	21	20	41
Item.java	70.4 %	19	8	27
panel	69.6 %	450	197	647
others	51.1 %	144	138	282
src/test/java	100.0 %	1,353	0	1,353
integration	100.0 %	699	0	699
unit	100.0 %	654	0	654

Figure 1. Distribution of test coverage

Findings (change in production code, bugs, and improvement in quality of code)

- **Finding 1: increased code reusability and testability**

The four private int methods for wall detection originally resided in the Panel class and returned an integer variable called “direction” that only the Panel has (Figure 2). However, as we created unit tests for the wall detection methods, we discovered that private methods generally are not tested, and if they are so complex that they need testing, the methods should deserve their own class. Therefore, we created a separate class called WallDetection that holds these four methods and changed them to boolean functions (Figure 3). By doing this, the code is more reusable because not only the Panel class can use the wall detection methods, other classes can also use them by creating a WallDetection object and calling the methods. This improved the quality of our code and made it easier to test wall detection.

```
    private int up_stop() {
        for(int i = 0; i < cell.getMap().length; i++){
            for(int j = 0; j < cell.getMap()[i].length; j++){

                if(cell.getMap()[player.getPlayerX()][player.getPlayerY() - 1] == 0) {
//                  System.out.println((player.getPlayerX()) + " " + (player.getPlayerY()-1));
                    direction = -1;;
                }
            }
        }
        return direction;
    }
```

Figure 2. Old version of wall detection: private int methods in Panel class.

```
public class WallDetection {

    private Cell cell;
    private Player player;

    public WallDetection() {
        cell = new Cell();
        player = new Player();
    }


    public boolean right_stop() {

                if(cell.getMap()[player.getPlayerX() + 1][player.getPlayerY()] == 0) {
                    return true;
                }
        return false;
    }
```

Figure 3. New version of wall detection: public boolean methods in

- **Finding 2: change in production code, fixed bugs**

A bug that we found is that when we were testing the detectWin() method, a test case kept on returning error even though the logic seemed correct and the game was working perfectly fine (Figure 4). One condition for Player to be still in game is when Player is on the exit cell but has not collected all the regular rewards. In this case, the string that holds the state of the game is supposed to return "GAME." Yet, the string returned "LOSE" in the unit test (Figure 4). We also made sure that Player and Moving Enemy did not collide but it was still returning "LOSE" instead of "GAME." As a result, we modified the code for detectWin() by changing the if-else statements. It fixed the issue in that the string is finally returning the correct state for all winning and losing conditions (Figure 5).

```
@Test
void testStillInGame() {
    int score = 10;

    // Player is still playing when on the exit cell
    // but has not collected all the regular rewards
    p.move(9, 7);
    state = cell.detectWin(score, 8, 15);
    assertEquals(state, "GAME");

}
```

Figure 4. Test case for testing if Player is still in game and returned the wrong value before modifying the if-else statements.

```
public String detectWin() {

    // check if the score is negative and change the game state to "LOSE" if it is
    if(score < 0) {
        Panel.stateStr = "LOSE";
    }

    // check if all regular rewards have been collected and if Player is on the exit cell
    // change the game state to "WIN"
    if(collected == 15 && Player.getPlayerX() == 9 && Player.getPlayerY() == 8) {
            System.out.println("Done");
            Panel.stateStr = "WIN";
    }

    return Panel.stateStr;
}
```

Figure 5. Old version of detectWin(). Returned "LOSE" instead of "GAME". Winning and losing conditions depend on class' private variables.

```
public String detectWin(int score, int collected, int rewardNum) {

    // check if the score is negative and change the game state to "LOSE" if it is
    if(score < 0) {
        Panel.stateStr = "LOSE";
    }

    // check if all regular rewards have been collected and if Player is on the exit cell
    // change the game state to "WIN"
    if(Player.getPlayerX() == 9 && Player.getPlayerY() == 8) {
        if(collected == rewardNum) {
            System.out.println("Done");
            Panel.stateStr = "WIN";
        }else {
            Panel.stateStr = "GAME";
        }
    }

    return Panel.stateStr;
}
```

Figure 6. New version of detectWin(). Fixed the testing issue. Winning and losing conditions now depend on parameters that are passed to the method.

- **Finding 3: increased code reusability and testability**

The third finding continues off the second finding. While modifying the if-else statements in detectWin() to carry out successful tests, we also added method parameters to detectWin() to make it less hard-coded, more reusable, and have better testability. Now, the losing condition (if score < 0) does not depend the private score variable in the Cell class (Figure 6), but rather an integer variable that is being passed to the method (Figure 6). In addition, the variable that keeps track of how many regular rewards that Player has collected does not depend on the class' private "collected" variable, but a "collected" variable that is also being passed to the method. Lastly, the number of regular rewards that Player has to collect to win

the game is not a constant that needs to be changed manually anymore (Figure 5), but an integer variable called “rewardNum” (Figure 6).

- **Finding 4: increased code reusability and testability**

The fourth finding is regarding the code for generating regular rewards its enhanced testability. We ended up modifying and improving the quality of our code because we wanted to test the reward generator feature. Originally, the code for randomly placing the rewards and punishments were in the Cell class’ constructor (Figure 7). As a result, the code was not reusable and it was impossible for us to test if the rewards and punishments were generated correctly for each game, including properties such as the number of these items generated and whether if they were placed on a valid cell. Therefore, we created a new method called rewardGenerator(int count, int total) solely for randomly placing the regular rewards on the map and returns an integer that specifies how many regular rewards have been generated, which is also the total number that Player has to collect to win (Figure 8). We also added two parameters to the method because it makes the algorithm more flexible and reusable. This allowed us to perform integration testing on the rewards and detectWin() because we can pass the returned integer value “count” from rewardGenerator() to the detectWin() method and use it as “rewardNum.”

```
public Cell() {
    cellSize = 60;
    score = 0;
    time = 0;

    for (int i = 0; rewardNum < 15; i++){
        randomX = (int)Item.random(1, 9);
        randomY = (int)Item.random(1, 9);

        // only add a regular reward to the board when the cell found is empty
        if(item_map[randomX][randomY] != 0 && item_map[randomX][randomY] != 99) {
            item_map[randomX][randomY] = 99;

            rewardNum+=1;

            // when an unused cell is found,
            // add a new reward to the reward arrayList with this position:
            items.add(new Reward (randomX, randomY));
        }
    }
}
```

Figure 7. Reward generator used to be in the Cell class’ constructor. This was inflexible and not reusable.

```
public int rewardGenerator(int count, int total) {
    // randomly generates the position for each regular reward,
    // and change the value of that cell to 99
    // to indicate that it contains a regular reward
    for (int i = 0; count < total; i++){
        randomX = (int)Item.random(1, 9);
        randomY = (int)Item.random(1, 9);

        // only add a regular reward to the board when the cell found is empty
        if(item_map[randomX][randomY] != 0 && item_map[randomX][randomY] != 99) {
            item_map[randomX][randomY] = 99;

            count+=1;

            // when an unused cell is found
            //add a new reward to the reward arrayList with this position:
            items.add(new Reward (randomX, randomY));
        }
    }

    return count;
}
```

Figure 8. New reward generator method. Added parameters and a return value so that it’s more reusable and gives better testability.