**Unit Tests**

* + - **Feature 1: Wall detection**
* Description: The first feature to be united tested is 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.

**Integration Tests**

* + - **Interaction 1: Number of rewards and winning conditions**
* Description: The second feature to be united tested is 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.

**Test Quality**

**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 #). 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 #). 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.

A screenshot of a cell phone

Description automatically generated

Figure #. Old version of wall detection:

private int methods in Panel class.

A screenshot of a social media post

Description automatically generated

Figure #. New version of wall detection:

public boolean methods in

WallDetection class.

* + - **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 #). 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 #). 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 #).

A screenshot of a cell phone

Description automatically generatedA screenshot of a cell phone

Description automatically generated

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

Figure #. Old version of detectWin(). Returned “LOSE” instead of “GAME”. Winning and losing conditions depend on class’ private variables.

A screenshot of a social media post

Description automatically generated

Figure #. 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 #), but rather an integer variable that is being passed to the method (Figure #). 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 #), but an integer variable called “rewardNum” (Figure #).

* + - **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. 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. 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.”

A screenshot of a social media post

Description automatically generatedA screenshot of a cell phone

Description automatically generated

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

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