**Player Config Tests**

1. Entering Invalid Player Name Gives Alert
2. Entering a null string in the text field for “Name” gives an error
3. Entering a string with only multiple empty spaces in the text field for “Name” gives an error
4. Part of M2 implementation requirements for the player configuration on the “Welcome” screen
5. Checking to make sure that name is valid because “” or “ “ or null or the input text description are not valid names for the player.
6. Player Starting Money is Different Based on Difficulty Levels
7. Difficulty Level of 1 allows the player to start with $2000
8. Difficulty Level of 2 allows the player to start with $1500
9. Difficulty Level of 3 allows the player to start with $1000
10. Checks to see that the starting money in the Player class is different when inputting different difficulty levels
11. Part of M2 implementation requirements for the player configuration on the “Welcome” screen
12. This makes sure that the player config takes the difficulty and implements the different money values correctly and that they are different.
13. Player Starting Health is Different Based on Difficulty Levels
14. Difficulty Level of 1 allows the player to start with 500 health
15. Difficulty Level of 2 allows the player to start with 400 health
16. Difficulty Level of 3 allows the player to start with 300 health
17. Checks to see that the starting health in the Base class is different when inputting different difficult levels
18. Part of M2 implementation requirements for the player configuration on the “Welcome” screen
19. This makes sure that the player config takes the difficulty and implements the different health values correctly and that they are different.
20. Entering null difficulty gives alert
    1. Causes method to return “Must choose a difficulty”
    2. Part of M2 implementation requirements for the player configuration on the “Welcome” screen
    3. Makes sure that a difficulty value of 1, 2, or 3 are received correctly. If the player does not input the difficulty level, it would make the rest of the game not function. This difficulty level determines the health, starting money, and prices of the towers.

**Shop Tests**

1. Player purchase tower successfully
   1. checkPurchaseTower1()
      1. Checks to see if the Tower 1 bought at difficulty 1 correctly decreases the player’s money by the tower’s price.
   2. checkPurchaseTower2()
      1. Checks to see if the Tower 2 bought at difficulty 1 correctly decreases the player’s money by the tower’s price.
   3. checkPurchaseTower3()
      1. Checks to see if the Tower 3 bought at difficulty 1 correctly decreases the player’s money by the tower’s price.
   4. These tests are important for us to check that the player money is decreased by the correct value.
2. Money remaining is different for same level different tower
   1. checkSameLevelDifTowerMoneyChange()
      1. Creates 3 towers of different types, and purchases them. Checks if each tower costs a different price by comparing the amount of money the player has left after purchasing.
   2. This test is important because it tells us that the type of tower affects the values of the tower. The result of the player money after the purchase for each one will be different.
3. Money remaining is different for different level same tower
   1. checkDifLevelSameTowerMoneyChange()
      1. Creates a Player that buys 3 towers, each at a different (upgraded) level. Checks whether the cost of each tower is different at each level.
   2. This test is important because it tells us that the level affects the values of the tower. The result of the player money after the purchase for each one will be different.

## 

## **Tower Place Tests**

1. Tower can not be placed onto the path
   1. The new tower cannot be placed onto the path
   2. Checks functionality:
      1. towerIsNotOnPathTest()
         1. Created an array of path rectangles and configured coordinates so that the new tower will not be in the same area as the path and tested if the method of checking isTowerOnPath results in False.
      2. towerIsOnPathTest()
         1. Created an array of path rectangles and configured coordinates so that the new tower will be on the same coordinates as the path and tested if the method of checking isTowerOnPath results in True.
   3. Important to check the cases of where the tower can be placed. Since it cannot be on the path, we can see if the boolean value resulted by checking if it is touching is true or false.
2. A tower can not be placed onto another tower
   1. If current towers null, new tower should always be placed
   2. The new tower cannot be touching with current towers
   3. Checks functionality:
      1. towerIsNotOnTowerTest()
         1. Created an array of already placed towers and configured coordinates so that towers would not be near each other and tested if the method of checking isTowerOnTower results in False.
      2. towerIsOnTowerTest()
         1. Created an array of already placed towers and configured coordinates so that the new tower will be in the same place as an already placed tower and tested if the method of checking isTowerOnTower results in True.
   4. Important to check the cases of where the tower can be placed. Since it cannot be on another tower, we can see if the boolean value resulted by checking if it is touching is true or false.

## **Base Tests**

1. Tests if Base has health left:
   1. testIsBaseHealthy()
      1. Checks if the method correctly returns false if base health drops to 0.
      2. If health is still above 0, it returns true.
      3. Makes sure that testIsBaseHealthy() functions correctly.

## **Game Start Tests**

1. Testing attackBase() method for each enemy
   1. testEnemy1AttackBase()
      1. Checks to see that Enemy1 decrements the health correctly when attacking the Base. Compares the DPS of the instance of Enemy1 to the difference in the health before and after the attack.
      2. Makes sure that the attackBase() method is implemented correctly
   2. testEnemy2AttackBase()
      1. Checks to see that Enemy1 decrements the health correctly when attacking the Base. Compares the DPS of the instance of Enemy1 to the difference in the health before and after the attack.
      2. Makes sure that the attackBase() method is implemented correctly
   3. testEnemy3AttackBase()
      1. Checks to see that Enemy1 decrements the health correctly when attacking the Base. Compares the DPS of the instance of Enemy1 to the difference in the health before and after the attack.
      2. Makes sure that the attackBase() method is implemented correctly.
   4. testEnemyAttackBaseDifferently()
      1. Each enemy is of a different level in terms of how hard it is to defeat. Therefore, each enemy has a different DPS value or damage per second.
      2. Checks to see that type of enemy affects the value the enemy attacks the base with, comparing the values of the DPS for each instantiation of a type of enemy.
2. testing createEnemy() method if create the right type of enemy with initial given position.
   1. test createEnemy1():
      1. create an enemy1 type Enemy1 with X = 1180 and Y = 270
      2. call the createEnemy(0) method to see if it returns the right enemy (enemy1test) with Enemy 1 type
      3. compare the enemy1 and enemy1test to see if it has the same XY-coordinates.
      4. Compares the classification of enemy1 and enemy1test, checking to see that they are the same
      5. Checks to see if the enemy1test’s classification is “Yellow” matching the classification name for each enemy1
   2. test createEnemy2():
      1. create an enemy2 type Enemy2 with X = 1180 and Y = 270
      2. call the createEnemy(1) method to see if it returns the right enemy (enemy2test) with Enemy 2 type
      3. compare the enemy2 and enemy2test to see if it has the same XY-coordinates.
      4. Compares the classification of enemy2 and enemy2test, checking to see that they are the same
      5. Checks to see if the enemy2test’s classification is “Green” matching the classification name for each enemy2
   3. test createEnemy2():
      1. create an enemy3 type Enemy3 with X = 1180 and Y = 270
      2. call the createEnemy(2) method to see if it returns the right enemy (enemy3test) with Enemy 3 type
      3. compare the enemy3 and enemy3test to see if it has the same XY-coordinates.
      4. Compares the classification of enemy3 and enemy3test, checking to see that they are the same
      5. Checks to see if the enemy3test’s classification is “Pink” matching the classification name for each enemy3
3. Each type of enemy should have different health.
   1. testEnemyHealth()
      1. Creates an enemy type 1
      2. Creates an enemy type 2
      3. Creates an enemy type 3
      4. compares their health and return true if their health are different
      5. checks to see that different enemy types result in different health values
4. Enemies are walking
   1. Testing that an enemy walks correctly when on a certain portion of the path

A picture containing icon

Description automatically generated

Upper Path

Lower Path

Middle Path

* + 1. testEnemyWalkUpperPath()
       1. Checks if the enemy on the upper path correctly walks to the left
       2. Creates an Enemy1 on the upper path
    2. testEnemyWalkMiddlePath()
       1. Checks if the enemy on the middle path correctly walks downwards
    3. testEnemyWalkLowerPath()
       1. Creates enemy1 on the lower path close to the base
       2. Checks if the enemy on the lower path correctly walks left into the base and disappears
    4. Initial x and y position of each enemy in the array list at the start of the game is different from their x and y position when they have walked to reach the base
  1. Testing that different enemies walk correctly when in the upper path
     1. testEnemy1IsWalking()
        1. Creates an ArrayList containing enemy1
        2. Calls the enemyWalk() function with the ArrayList created as a parameter
        3. Checks to see that the change in x from its initial position to its position after walking is its walking speed.
     2. testEnemy2IsWalking()
        1. Creates an ArrayList containing enemy2
        2. Calls the enemyWalk() function with the ArrayList created as a parameter
        3. Checks to see that the change in x from its initial position to its position after walking is its walking speed.
     3. testEnemy3IsWalking()
        1. Creates an ArrayList containing enemy3
        2. Calls the enemyWalk() function with the ArrayList created as a parameter
        3. Checks to see that the change in x from its initial position to its position after walking is its walking speed.
     4. Makes sure that the enemyWalk() method correctly functions for each enemy, taking into account their different walking speeds

**Enemy Tests**

1. Checks to see if Tower1, Tower2 and Tower3 are in proximity to Enemy1
   * + - 1. testEnemy1Proximity()

Creates a new Enemy1, Tower1, Tower2 and Tower3

Asserts false if Enemy1 is not in proximity and true otherwise

1. Checks to see if Tower1, Tower2 and Tower3 are in proximity to Enemy2

testEnemy2Proximity()

Creates a new Enemy2, Tower1, Tower2 and Tower3

Asserts false if Enemy2 is not in proximity and true otherwise

1. Checks to see if Tower1, Tower2 and Tower3 are in proximity to Enemy3
   * + - 1. testEnemy3Proximity()

Creates a new Enemy3, Tower1, Tower2 and Tower3

Asserts false if Enemy3 is not in proximity and true otherwise

## **Tower Tests:**

1. Tests if the distance calculator gives the right result:
   1. distCalculatorTest()
      1. Checks if the method correctly returns true if the distance between tower and the enemy is equal.
      2. If the two distance are not equal then return false.
2. Tests if the closestEnemy method correctly outputs the enemy from an arraylist of enemies with the shortest distance from the current tower instance.
   1. testTower1ClosestEnemy()
      1. Checks if the method correctly returns the Tower1’s closest enemy from an arraylist of enemies with different positions.
      2. Adds another enemy that is closer to the Tower1 after checking the original arraylist to make sure that the method is valid
      3. Important because it is used to determine which enemy to attack.
   2. testTower2ClosestEnemy()
      1. Checks if the method correctly returns the Tower2’s closest enemy from an arraylist of enemies with different positions.
      2. Adds another enemy that is closer to the Tower2 after checking the original arraylist to make sure that the method is valid
      3. Important because it is used to determine which enemy to attack.
   3. testTower3ClosestEnemy()
      1. Checks if the method correctly returns the Tower3’s closest enemy from an arraylist of enemies with different positions.
      2. Adds another enemy that is closer to the Tower3 after checking the original arraylist to make sure that the method is valid
      3. Important because it is used to determine which enemy to attack.