M3 JUnits

1) configSpaceName(): This test checks if the program correctly identifies a valid player name that includes characters besides only whitespace. If a player name is an empty string, the configError() method returns null. In the test case, name is initialized to one whitespace character which configError() will trim and evaluate as a null String. If Config.configError(name, true)'s value is null, it matches up with the expected value. The test passes only when the program correctly identifies the test case of a whitespace character as an invalid player name. This test is relevant to the implementation requirements because a player must have a valid name in order to continue to the game.

2) configNoName(): This test checks to see if the program detects a player with no name. An empty String passed into the test is an invalid player name, and the program will accordingly output a null in this case. A null is to be expected so the test case passes on the condition that the program correctly checks that there is not a player name. This test is relevant to the implementation requirements because players must have a name with characters other than only whitespace in order to be able to play the game.

3) configNoDifficulty(): This test checks if the program does not detect a difficulty level. When there is no difficulty level, false is passed into Config.configError(name, false) which accordingly returns a "noDifficulty" from the lack of difficulty level chosen. This matches the expected value of "noDifficulty" resulting in a passing test which confirms that the program correctly identifies when a difficulty level has not been determined. This test is relevant to the implementation requirements because the money, monument health, and purchasing costs of towers all depend on the type of difficulty chosen. Without a chosen difficulty, the program will not be able to set up the game.

4) configWorks(): This test represents the scenario where the configuration screen has a valid name and chosen difficulty level. In this case, the configuration screen should accept these values and set up the game with the proper money value, monument health level, and purchasing cost values. When Config.configError(name, true) outputs the name, this confirms that the test passed with an actual player name with characters other than only whitespace. This test is relevant to the implementation requirements because a functional configuration screen should allow the player to continue to the game only when the player name is valid and a difficulty level has been determined. The game can adequately set up the money, monument health, and purchasing costs of towers after knowing the difficulty level.

5) testIsSufficientFundForEasyTower1(): This test checks if the program correctly allows the purchase of a tower when player money is greater than the tower cost. The test specifically determines the cost for tower 1 on the easy level as $32 and with playerMoney set to $63, the program should increment tower count to indicate a purchase. In this case, the program returns the tower cost ($32) and the test expects $32. The test fails when the program does not correctly determine that playerMoney is not greater than the cost of the tower. This test is relevant to the implementation requirements because towers must only be purchased with

sufficient funds. Successful purchase of a tower (represented by an increment of tower count) is crucial to a player's defense in the game.

6) testIsSufficientFundForEasyTower2(): This test checks if the program bars a player from purchasing towers due to insufficient funds. In this scenario, the test case simulates a player with $30 attempting to purchase tower 1 on the easy level ($32). Since the playerMoney is less than the cost of the tower, the program should return 0 and not increment tower count. This test is relevant to the implementation requirements because tower count must not increment when insufficient funds are used in an attempt to purchase a tower.

7) notEnoughTowersToPlace(): This test simulates a scenario where the program detects that there are not enough towers in inventory when a place tower is attempted. By instantiating a TowerList variable with the cost of 32, we are stating that there exists a tower type (in this case, it is an easy tower1). Then we are attempting to place the tower on the map. Since there were no towers purchased, there are no towers available in inventory. placeTower() will check inventory and will return false since there are no towers in inventory. This test is relevant to the implementation requirements because when there aren't enough towers in inventory, towers that a player has not purchased should not be able to be placed.

8) enoughTowersToPlace(): This test simulates a scenario where there are enough towers to place a tower. By instantiating a TowerList variable with the cost of 32, we are stating that there exists a tower type (in this case, it is an easy tower1). The user is able to purchase 1 tower since they have enough money and the inventory size will increase by 1. When placeTower() is called, it returns true as there is at least 1 tower in inventory. The assertTrue statement will also check if the inventory is 0 as placeTower() removes the tower from inventory. This test is relevant to the implementation requirements because the player should be able to place towers if there are enough towers in inventory. Proper functionality of this feature is important for the player's defense in the game.

9) towerPricePerDifficulty(): This test checks if the program correctly differentiates in the tower prices depending on the difficulty level. Tower 1 on easy level should be the lowest cost followed by tower 1 on the medium level. Tower 3 on the difficult level should have the highest cost. When this test passes, this indicates that the program successfully identified a different cost depending on the level for each tower. This test is relevant to the implementation of the program because tower costs must vary accordingly with difficulty level determined and the type of tower selected as well. Varying tower costs are crucial to the game's complexity.

10) correctInventory(): This test checks if the program correctly represents the right inventory amount after the purchase of a tower. Tower count should increment by one after each purchase with sufficient funds, and the test case checks this by purchasing two towers with $300 in playerMoney. The tower cost that the program is checking for purchase is a $32 tower and by purchasing two towers, inventory must show a tower count of 2. The test passes when tower count is 2 and fails when inventory shows a tower count of a number other than 2. This test is relevant to the implementation of the program because placing towers relies on the tower count

in the inventory to function correctly so the player can place only the exact number of towers they have purchased from the shop.