CMPT 370 Testing Plan Document for BattleBots

**Group A5**

Mackenzie Power

Will Revell

Haotian Ma

Ryan Tetland

Mitchel Kovacs

Phase3\_TESTING

Table of Contents

Introduction ……………………………………………………………...03

Summary ………………………………………………………………….

Model Testing:

The model in our architecture consists of several object classes that interact with each other through the controller in various ways. The conglomerate of classes that comprise the model are: GameBoard, Hex, Robot, Scout, Sniper, Tank, ScoutAI, SniperAI, TankAI, Gang, and GangAI, Many of them have attributes of other classes within the model. Many of these classes can be tested using J-Unit tests to ensure correct functionality. Since our main Integrated Development Environment (IDE) used to create the program is NetBeans, it has a function that is able to generate setter, getters, and J-Unit testes for each function that is used in a given class. Due to time constraints, we will not use create J-Unit tests for any getter and setter functions. Since the IDE is able to automatically generate those functions, we can assume safely that they should be able to work. Testing for these functions may be considered if we become ahead of schedule during implementation. However, these functions will be important for the testing of other functions that will result in changes of attributes of other classes.

GameBoard class:

The simplest functions to test for in the game will be those that are associated with the GameBoard class and Hex class. As discussed, simple J-Unit tests can be used to determine if these classes are able to be constructed correctly. With the GameBoard class, we will have to make sure the J-Unit tests check to see if the board is able to instantiate with the correct size and number of Hex spaces (when given a specific integer). This will be done by asserting within the test that all playable spaces exist (are not equal to null), and spaces that are not used will be set to null (since our game board is a 2D array of Hex spaces). For the Hex spaces, it has only the one function which is to check to see if the Hex List of robots it contains is empty. Since the function simply returns one of three different String values, the J-Unit test will assert a specific string depending on if there is a robot present or if the Hex is out of bounds in the game. For this test and many others, we will have to create mock objects of the Robot class to test the functions that involve use of robots. Within the J-Unit test, we will have to instantiate a new robot and then hard code it into the Hex object list and check that if can return a correct value.

Robot class:

There will technically be no objects in our program that will of the Robot class. There will, however, be many objects of classes that inherit from the Robot class. All of those classes will continue to utilize most of the functions from the Robot class. As mentioned previously, most of the functions involved in these classes are setter and getter functions and will therefore not be tested individually. We will, however, perform a J-Unit test for the constructor of each of the following classes: Scout, ScoutAI, Tank, TankAI, Sniper, and SniperAI. Since we already know what the various attributes of each robot should be (range, health, damage) we will construct each Robot class and determine whether or not each constructor can successfully set the various parameters correctly, since each child class has a new constructor that is redefined. These J-Unit tests will use the assertEquals functions in Netbeans to check that each and every attribute has been accurately assigned based on what we know the attributes should be.

The receive damage function should be able to take away a specified amount of damage from robots that call the function. J-Unit tests for that function will make sure that the appropriate amount of heal is subtracted from each robot and test that robot health cannot go below zero. During turn moves and scans, alive robots that are in play will be determined by having health that is greater than zero. An exception should be caught whenever the function is called on a robot with no heath since we need to consider the possibility of a Hex space being attacked that contains both and alive robot and a dead one.

The AI robots (ScoutAI, TankAI, and SnipreAI) all have the additional functions of move(), turn(), scan(), and shoot(). Since these functions require the use of mock robot classes to test their ability to function correctly, we will have to create mock game simulation to ensure that those functions work appropriately.

Since the GameBoard class will always be created and used in every game and simulation, we will create all tests that involve interaction between model classes here as a testing main function. Note: for all exceptions that will be thrown for these series of tests, each new exception thrown due to expected values not matching resulting values will have an accompanying message that prints out a detailed message of what the two compared values were and what test was failed. Robots AIs must be able to move, shoot, scan, and turn. We will have to be realistic with our testing because if we wanted to test every conceivable action and move at every position then we would likely be creating tests for years. Since we have only a month for coding, we will have to only test basic scenarios for each function and the boundary conditions for them as well. To do this, we will have to hard code in a couple of mock robot AIs (two of each type) and assemble them into two different GangAIs. From here, we will initialize a new GameBoard and code them on opposite sides of the board. From here, we will be able to test execution of the various functions the robots employ to play the game. Note:We will then execute the move function for each robot, attempting to move them to different parts of the game board, including spaces that go beyond the boundaries of the board. Since moving outside the board will throw an exception that is caught within the move function, we should test using asserts that the position of the robot remains unchanged when attempting to move to these positions. Since every robot retains the previous position it was in we will have to test that is has changed. In addition, we will have to test that the robot remains in its current position if all of its moves are used up. These tests can be done by throwing exceptions when the value obtained does not match the expected value.

The scan function will return word values based on whether or not there are any robots present in a scanned Hex space. These series of tests will involve moving the robots around to different spots on the board and see if the scan function can correctly return the information. We will have to test that it will return correct values for when there are no robots present; one robot present; two robots present; one single dead robot; and also if there are dead and alive robots present. If the returned values are not equal to the values we expect then exceptions will be thrown.

Testing the turn function should be a relatively straight forward. Simple J-Unit test would suffice, but since it is part of every move a robot AI makes we will include it into these series of tests. Since each robot will have an attribute known as the relativeDirection which (as the name implies) contains the robots pointing direction, all we need to do is check that the value is incremented by one each time scan is performed. If it is not then an exception will be thrown.

The final function which will be needed for this series of tests will be the attack function. We must test that a robot will not fire upon a Hex space that contains either no robot or only dead robots. For this we will have to compare the shotsLeft attribute on the robot to ensure that no shots were allowed to fire. If they were, then an exception will be thrown. Next, for each robot type, we must ensure that they inflict the correct amount of damage. Test this, we will have robots call their attack function on a robot in range and then compare the shotsLeft attribute and the health attribute of the robot being attacked then throw an exception if the values are on what is expected. We will then need perform this on attacking Hex spaces that have more than one active robot; and also on spaces that have both an active and dead robot on them and compare those values.

Changes:

Each robot will have a reference to the GameBoard.

Add a receive damage function for each robot that takes in an integer (attackers damage).

Move and attack functions for human controlled robots