# Testing Document

Brady Lang

Dylan Mcinnes

Parker Reese

Meagan Renneberg

***Legend:***

Underline for Entities

**Bold** for **methods**

*Italics* for *non-method members*

***Intro***

The testing of our program will be done using Test Driven Development. Test Driven Development is a type of Unit Testing done before any program code has been written. Given the requirements of our methods and entities, specific test cases are created which our software needs to pass. The software is then incrementally improved to pass all the test cases.

This type of testing is beneficial because it forces - at minimum - an operational program. After first completing our significant interfaces we can move on to completing the less significant interfaces. By doing the testing incrementally we negate the possibility of being unsure as to what part of our code is no longer working, for we would have adequately tested everything previous to the most recent addition.

All of our tests will be located in the main() method of each entity. This splits up the testing for each entity locally to itself. By using asserts for each test case we will eliminate the possibility of unknowingly assuming a test case has passed. If a test case has failed, the assert will log itself as having failed, and will exit the execution of the program.

Test Driven Development has certain advantages over other testing methodologies such as System Testing and Integration Testing. Namely, Integration Testing, which is the testing of individual software modules combined, is redundant. This is because - given our individual entities which have all passed their unit tests - our mock-inputs and inputs coming from another entity will be identical. If the only test we performed was a System Test, it would create a possible plethora of errors which would be

exceedingly difficult to debug all at once. Since System Testing is simply the testing of the whole system at once, it would be an enormous assumption for one to make that the whole system works without any testing prior. Thus, given the size and scope of this project, Test Driven Development not only works, but will increase guaranteed functionality of the program.

***Interfaces and Tests***

Forth Interpreter

The ForthInterpreter takes a Script and translates the logic in this Script into Java method calls. These Java calls (in the actual program) consist of actions to be made by the NPC Robot during the game. Such actions may be move, or shoot, as well as where it should move and or shoot. Without proper behaviour from this entity our NPC Robots will not act, re-act, or communicate with its Team, in any useful fashion. We consider users having an interactive experience with enemy NPC Robots to be an important factor in the game's playability.

We will be testing the ForthInterpreter's ability to correctly call the Forth Words in the correct order, and use proper stack behavior to handle "arguments" for the various Words. To do this we will have multiple Scripts, each with their own different logic/ordering. Given these different scripts, the ForthInterpreter should have an appropriate output. We will also define other Forth Words which will call Java methods to assist with testing.

First, we will test that the ForthInterpreter can execute a static, linear series of calls in the correct order. Second, we will test the execution of a structure that contains a conditional branch. Third, we will test the interpreter's ability to get and set variables. Fourth, we will test a more complex structure with complex Words (ie. Words that are composed of two or more words executed in a linear fashion).

The correct behavior of all of these tests will be determined by running through the Forth code by hand and checking the proper ordering of “Java” calls - which are not actually Java code while running them by hand.

Board Position

The BoardPosition controls all aspects of distance and direction that is related to each robot, as well as the distance and direction relative to every robot. It is important for BoardPosition to be accurate because if it does not work the robots will both be moving and interacting incorrectly.

To assure that the robots are displaying the correct behavior, we will test the functions **turn** and **advance** by setting up a robot on a specific spot. The robot is then moved one space and the test checks to make sure that the robot’s coordinates are not the same as the starting coordinates and the coordinates are now equal to what the expected coordinates are (eg. starting coordinates are (0,1) and robot is moved one space to the right so expected coordinates should now equal (0,2)). This test will also check that the robot turns in the correct direction when moving. It checks that if the space it is moving to is not in the direction it is facing, it turns to face the new direction of the new space. So the test will check if starting direction is different from current direction. (eg. Facing direction is always 0 so if turned to face direction 4, check that direction 4 now is equal to 0).

To test **immediateDirection,** two robots are set so the robots are not on the same space. When called **immediateDirection** should return the neighboring direction to get from robot 1 to robot 2, For example, if robot 1 is in the direct centre of the board and robot 2 is on a space that is labeled 15 on figure 1. The

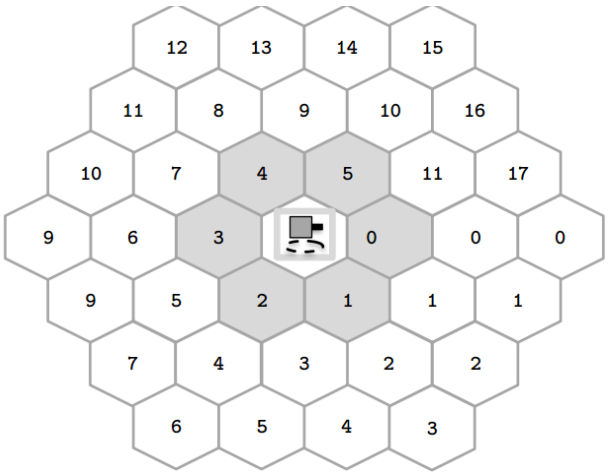


Figure 1

function should return 5. If robot 2 is on space 14, the function should return either 4 or 5 as a valid answer.

To test **directionTo,** and **distanceTo** a single robot is set on the board. When called, **distanceTo** should return the the distance between the robot's current position and the other position. The function **directionTo** works similarly to **distanceTo** but instead returns the direction that the robot must turn to in order to get to the other position. For example using figure 1 as reference, set the robot in the centre of the board. The functions test the direction and distance from the robot's position to the space labeled 17. The function **distanceTo** should return 3 and the function **directionTo** should return 17.

To test **getRelative,** a single robot is set on the board. When called and given a distance and direction it should return a new BoardPosition related to the given inputs. For example if the robot is placed in the centre of the board as seen in figure 1, and **getRelative** is given the inputs of direction of 4 and distance of 1, it should return the BoardPosition that corresponds to the space that is direction 4 and 1 space away.

GameBoard

The GameBoard is a container of information pertaining to the game. GameBoard retains statistics based on current events and determines what is and isn’t a valid spot.

This interface is significant because it is important for there to be constraints with our game board in terms of where robots can move and shoot. The game will perform incorrectly if a robot is able to interact with an invalid space. It is also significant because the GameBoard needs to retain the correct statistics. If the statistics are being wrongly recorded then the statistics sent to the Robot Library will be incorrect.

We will be testing that players can not move or shoot into a space outside of the game board using the function **isValid**. The tests for the GameBoard requires some of the other entities to be correctly

implemented. For instance, the **isValid** method tests requires that the BoardPosition entity is correctly implemented. We will test that when given two map positions - one that is on the GameBoard and one that is not on the GameBoard - **isValid** returns the correct value. This is important for the methods **move** and **shoot** so we do not allow a robot to move off the game board. The method **updateImmediateStatistics** is tasked with updating the statistics of a particular robot script, with the statistics being: wins, losses, lived, died, absorbed, killed, and distance. The tests for **updateImmediateStatistics** will hard-code an update to a particular robot's particular statistic and then check to make sure that the correct statistic was updated with the correct number.

Referee

The Referee Interface is what oversees and manages our game as it progresses. It is one of our most involved controllers. It not only is in charge of managing the GameBoard but also the teams playing, the order of play, and updating the game board view.

If the Referee's behaviour is incorrect, the flow of the game would be inconsistent or invalid. The Referee dictates which robots turn it is, if the wrong robot is given the ability to play it would cause confusion. Also, when a team decides to quit, should the Referee incorrectly quit a different team, or not remove the team, the game would become unfair and have a negative impact on playability.

The tests for **teamQuit** will quit a team and check the following conditions: if there is one fewer team, if all three robots from the quitting team are dead, if there were only two teams did the non quitting team win, and if there were more than two teams did the game continue.

Since **startGame** calls the **updateImmediateStatistics** method in the GameBoard entity, we will test that it is being called with the proper arguments at the appropriate times. We will first test it by moving the robot to a valid spot and testing that the "distance" statistic is incremented. Then we will test it by dropping a robot's health to 0 and checking that the "win" statistic is false, and the "died"

statistic is incremented. Next we will test damaging a robot and checking that the “absorbed” statistic is increased by the same amount as the damage it took. Then we will force a robot to kill another and check that the “killed” statistic has been incremented for the killer. Finally we will end the game and check that all Robots that were alive had their “wins” condition set to 1, and all losing Robots had their “losses” statistic incremented.

Game Initializer

The Game Initializer interface controls what is initially displayed on the game board when the game is started. It takes in what was chosen by the user in terms of Teams and Robots and creates the corresponding GameBoard according to these decision. Thus, all Robots on a Team should be assigned the same color, be placed in the correct starting position, and a GameBoard of the correct size should be created.

This interface is significant for without a proper and consistent starting to the game, it may be deemed unplayable. Consider Robots of different Teams starting on the same position, or Robots having no identifiable Team for they are all randomly colored. This would not only create confusion, but also frustration for the users.

The test for **initializeGame** will check if all Robots have the correct location and direction, if all chosen Teams are in the game, and if the game board is the correct size. This will be done by supplying the GameInitializer with the appropriate information that will demonstrates its abilities to properly handle all of these cases. By “choosing” to have two and three teams should initialize a GameBoard of size five, and “choosing” to have six Teams should initialize a GameBoard of size seven. During these three different game initializations we will also be testing to make sure all the Robots and Teams are in their appropriate location and direction.

Robot Controller

The Robot Controller Interface is the main controller for both our human players and non-human players. Given the requests from NpcController and PcController, this interface manages the actions of the robots in the game.

This interface is significant because it dictates the players abilities (both human and non-human) to interact with their robots. Should the Robot Controller be unresponsive, there will no longer be any actions from the Robots in the game, and thus, making the game no longer playable. Of the methods in the Robot Controller entity, we will be testing **Scan**, **Shoot**, as well as **Send**. These methods are more easily testing once BoardPosition is functionally working and should be done after the entity is completed for optimal results.

The testing for **Scan** will consist of hard coding Robots at specific locations in respect to the origin of the scan location. **Scan** should, depending on the range of the Robot **Scan** is being used on, either scans a radius of 1, 2, or 3 hexagons. Thus, the hard-coded Robots will be placed on, within, and outside this radius. As well as boundaries of the range, we will also be sure that **Scan** can handle more than one Robot within its range. **Scan** should then return the list of Robots within its range.

The testing for **Shoot** will be done much the same way as **Scan**. **Shoot** should only damage another Robot if the position on the board matches the position that was shot. Also, depending on the Robot, you should only be allocated the ability to Shoot within a radius of 1, 2, or 3 hexagons. Also, depending on the Robots type, **Shoot** should cause 1, 2, or 3 damage to the Robots. To accomplish this testing, it is necessary for each type of Robot (Scout, Tank, Sniper) to be used to assure that damage is being dealt correctly. Also, a mock Robot should be present on, within, and outside of each Robot's range to test that the range a Robot is able to shoot is working correctly. Finally, **Shoot** should be directed to

positions without a Robot, with more than one Robot, as well as the position of the shooter itself, to finalize all possible conditions in our test cases.

The testing for **Send** will only need to accomplish two things. First, a message is sent to the correct Robot(s). Second, the message sent to the mailbox contains the appropriate information. The first condition will be accomplished by having a Robot send a message while there are both mock teammate Robots as well as mock enemy Robots present. **Send** should only put a message in the mailbox of teammates. The second condition will be tested in a number of ways. Depending on the intelligence of the Script, we may want it to send a message when it sees an enemy Robot, takes damage, gives damage, or kills a Robot. Thus, a message with each one of these scenarios must be able to be properly sent to other Robots with correct corresponding positions on the board.

Robot Library Socket

The Robot Library Socket Interface handles all communication between the robot library and our system. All interactions are done through JSON files and files in the proper format that can be transformed into JSON files.

This interface is significant because we need the robot library in order for our game to work with non-human players. If we can not communicate with the library to obtain AI scripts we will not have the ability to have functioning NPC Robots in our game.

The **updateStatistics** method is charged with receiving the intermediate statistics from the Gameboard and converting the data into a properly formatted JSON file to be given to the Robot Library. This JSON file must comply with the Robot Library's standards for format for it to properly change its documented statistics.

The testing of this conversion will be done by passing a mock Gameboard, with hardcoded immediate statistics, to the method, and having the JSON file compared to a pre-made JSON file with

the expected output. The hardcoded immediate statistics must consist of all types of different statistics (e.g. damage given, wins, loses), with different amounts of Robots, as well as different combinations of those statistics.

***Things We've changed:***

Changes we've made from our design document are that the ForthInterpreter 's variable *words* is now type Map<String, Word> instead of List<Word> because Words need to be String indexed. We also change the ForthInterpreter 's **exec(**Word**)** method to now be **exec(**String**)** so that the interpreter can execute "play" without needing to know what the sequence that "play" invokes is. The final change we have made is that the RobotController 's **scan** method should have a return type of List<Robot>, it initially was not returning anything but now returns the list of all Robots within range.

***Things to be Further Explained:***

The GameBoard 's hexes are either stored as a series of array positions, or as a graph of Hex nodes which are connected.. In either case, the structure itself is not directly accessible by any other entity in the system (with the possible exception of the BoardPosition entity), and as such does not interact with any of the other important interfaces. The positions on the GameBoard are deliberately abstracted through the use of the BoardPosition entity.

The BoardPosition entity has some *private* fields which describe its position and direction on a hex grid. This may be either a direction and array positions, or a Hex and a direction. Since no other entities directly access this information (with the possible exception of the GameBoard entity), it is not important to define how this is being accomplished until code construction.

The logical conclusion of the two above explanations is that both the GameBoard and the BoardPosition entities must be developed together.