Testing Document
CMPT 370

Group C4
Jack Huang
Brandon Jamieson
Ixabat Lahiji
Daniel Morris
Kevin Noonan

Intro

Diagrams

Interface x
- -Description
- -Specification
- -Testing plan

Interface x
- -Description
- -Specification
- -Testing plan

Interface x
- -Description
- -Specification
- -Testing plan

Changes
Summary

Board

Description:

In our design, the board component contains the state of the game and robot positions. Hence, our testing plan will require an interface to test our model's board class to ensure correct functionality. To accomplish this, the interface will need a "fake" robot object, as well as a "fake" hex object. The robot object will be necessary for moving around the board and ensuring invalid operations are handled. The hex object will be necessary for testing construction and functionality of the board itself. Making use of these objects, we can verify that our board performs all operations properly, as well as handling all invalid operations in a logical way.

Serving as a basis for the game, as well as a major component of the model, the board will be subject to numerous changes made during gameplay. As such, the board could be considered a "hotspot" in our system. If errors were to occur in the board, it could lead to robots being improperly displayed, attack targets not being correctly selected, as well as a plethora of other possibilities. Therefore, it is imperative that the functionality of the board is both proper, and consistent.

Testing begins by first initializing a board and verifying the correctness of its construction. Once this is confirmed, testing can take place on the operations of the board itself. We will need to verify that we can properly add or remove robots from hexes, which will occur whenever a robot moves or is spawned at the beginning of the game. As well, we will also need functionality for searching the board for robots based on a given hex and range, or the current robot playing. This will be used when a robot is attacking and will be needed for properly notifying the user of their possible attack options. We will also need to verify that we can damage hexes, applying the given damage to all robots within that hex, which will also take place when a robot is attacking. Finally, we will need to ensure that turns are properly transitioned, so that we can keep track of the current player's turn.

Unit Testing:

getHex(): The getHex method will be tested by first initializing a test board of size 5. We will then call the getHex method on varying positions on the board and check if it actually returns the hex that was placed in that position.

search(): The search method will be tested by first initializing a test board of size 5 and placing robots on the board. The method will then be tested multiple times to search varying positions on the board that either have 1 or more robots in the search area, or have no robots in the search area, the specified range is <=0, or the hex is not on the board.

addHexOcc(): The addHexOcc() method will be tested by first initializing a test hex and 2 test robots. The method will be tested by first adding one robot to the hex, adding the other robot to the hex, then adding the same robot again to the hex. The test will check the

listOfOccupants for the hex after each add, and will check that each robot has been added and that robots cannot be added to a hex if they are already in it.

removeHexOcc(): The removeHexOcc() method will be tested using the hex defined in the previous test and another defined test robot. The function will be called for each robot in the hex, a robot that is not on the hex, a robot that was previously on the hex, and when the hex is empty. The listOfRobots will be checked after each test to verify that correct behaviour is being met.

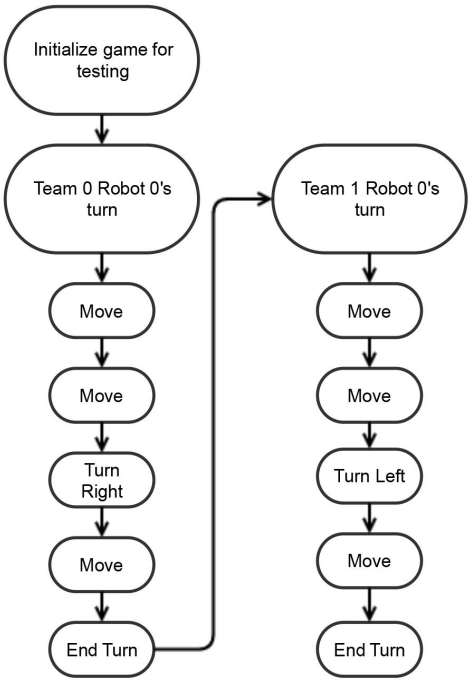
damageHex(): The damageHex() method will be tested by first damaging the previously defined hex with no robots in it. To find the damage stat of the current robot a robotTeam will have to be created and placed into the listOfRobotTeams. This robot team can be defined with a previously defined test robot in it with damage of 1. It will then add a previously defined test robot to the hex and call damageHex() again. It will then set the current robot to another test robot, and call damageHex() again. The listOfRobots will be iterated through after every test to make sure each robot has the correct health.

getTargetList(): The getTargetList() method will be tested by first defining numerous test robots and assigning them to different teams. The test will then change the currentRobot and currentTeam numerous times, calling getTargetList() each time and checking that the list is correct.

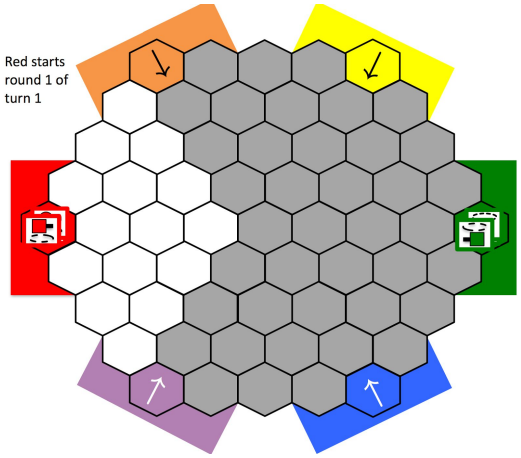
nextRobot(): The nextRobot() method will change the currentRobot and currentTeam numerous times, calling nextRobot() each time and checking that the currentRobot is being changed accordingly.

prevRobot(): The prevRobot() method will be tested in the same way that nextRobot() is tested.

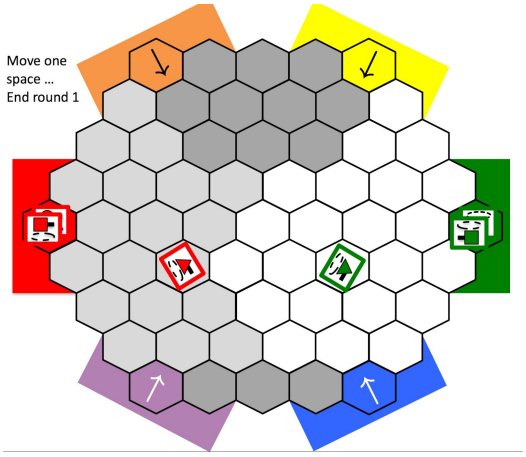
Moving Scenario:



Map of the board at the start of the scenario:



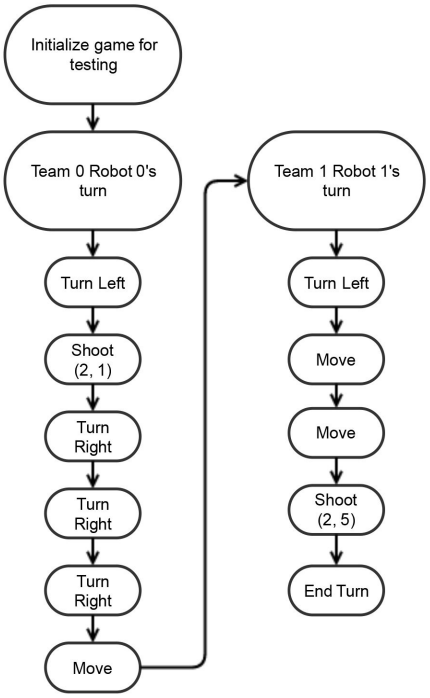
Map of the board at the end of the scenario:



To ensure that the board class is fully capable of handling multiple in a game like scenario we have devised this scenario to do so. The scenario starts with a two team game starting with team zero, the red team in this case. The red scout moves two spaces forward, turns to the right, then moves forward once again to end his turn. The green scout from team 1 moves two spaces forward, turns to the left, then moves forward once again to end his turn. The scenario ends.

This scenario will test all of the main aspects of the board excluding the functionality for selectedHex and the shooting functionality. The shooting functionality will be covered in the next scenario.

Shooting Scenario:



Robot Interpreter

Our testing plan will require a specialized interface for our robot interpreter. This interface will perform testing on our system's robot interpreter. As such, all testing will revolve around manipulating a robot object and ensuring that all possible operations are correctly handled. To check these, we can use a "fake" robot object with the same basic functionality as an actual robot, and verify that it behaves as intended when commands are issued. Both system-defined words and user-defined must be tested and verified to be in working condition. Since we cannot test every possible user-defined word, we will have to make sure we test a wide variety of them, each with different functionality to cover a wide range of situations.

While the purpose of this interface is simply to test the interpreter, its implications are much greater in the grand scheme of things. The interpreter is a key component in our system, handling both user and AI input to the robots themselves. Since the game revolves around the robots and their interactions with each other, it is crucial this component is thoroughly tested to ensure proper functionality. Any bugs in this component will derail the rest of gameplay, so we must know such things will not happen.

Testing Plan:

Since this interpreter interacts with robots directly, we will need a “fake” robot object with the basic functionality of a normal robot. This way, we can execute words and verify their correctness by checking the state of the robot, the stack, or both if necessary.

Since each robot will have their behavior script parsed at the beginning of the game, we will begin our testing by parsing a basic behavior script containing a range of user-defined words to also test. To verify that the parse worked, we can simply check that the robot has a valid play() function defined, and that all the user-defined words are in our variables list. Since we will know what words we are feeding it, confirming those words are in the system will be easy.

After this is done, we will need to verify that every word (both system-defined and user-defined) is functional. To test system-defined words, we will need to execute each one and confirm that either the robot, the stack, or both has the proper changes made. As such, testing will be performed in two sections: stack-manipulating words first, then robot-manipulating words.

For stack-manipulating words, we will need to ensure two things. One, any word that requires input from the stack should perform correctly when given such input, and assertions should be made if proper input is not given. Two, each word should output the correct value(s) to the stack when performed. Once these two properties are ensured, we can move on to the second section of testing.

Robot-manipulation words will involve some change in the robot’s state, as well as a possible change in the stack. However, before these words can be executed, the robot’s play() function must be called. Since we tested for a valid play function in our parsing, it should simply be a matter of executing play and verifying that the robot can now be manipulated.

For testing of these words to be correct, the two conditions for stack-manipulating words will need to be met, as well as a third condition: the “fake” robot will need to be properly manipulated when told to and assertions should be made if impossible commands are given to it. For example, the “move” word tells the robot to move one hex forwards in the current direction. To verify correctness of this move word, we would need to execute it and check that our “fake” robot has indeed moved forwards. However, since it is not possible for our robots to move out of bounds, the test should also ensure that if a robot attempts to move out of bounds, the action is properly handled and the robot does not perform invalid operations.

Controller

Description:

Testing Plan:

The controller is the only way for the view and the model to interact, so to test we need to create a “fake” user object that mimic what an user would do. When testing methods between the

two, we would have to create some new testing variable in both the view and the model. First we would set all the test variables to be false, then after we ran the method that we are testing, check if the test variable in the area that the method is suppose to effect have been changed to true.

For testing method that would change the game screen, we would check the setVisible variable of each components that is suppose to appear in the next screen, to see if it changed to true, and the last screen's components to be false. When testing the spectator's method, for S_pauseGame(), S_fastForwardGame(), and S_fogofWar we would check the different in the time variable and true or false of the return value from getFogofWar() method inside of the Spectator class

When testing method in the play game menu screen, we can test all the selection type of method the same way. Selecting the first variable in the list, then check in the model to see if the changes have been made. For P_setTeamName(), we would insert a test name into the text field mimicking the user input, then check the team name variable in the Robot Team class to see if it match the user input.

To test the option menu methods, we will check

Unit Testing:

The currentTurn() method would be tested by physically setting the current Team and current Robot variable an arbitrary number, then running the code to check if the number matches the one from current Team and current Robot.

The updateBoard() method would be tested by creating multiple test boolean variable for each method of the Board class, and the same variable inside of the InGame_Menu class. First set all the variable inside the Board class to be false, then inside of the reDraw() method of the inGame_Menu class, set all the test variable inside InGame class to be equal to the variable inside Board class. Now run the updateBoard() method and check if the variable inside the inGame class is true.

The updateUI() method would be tested the same way the updateBoard() method is tested, creating an test boolean variable inside of the Robot class, and the same inside of the InGame_Menu class, run the test and check if the variable matches.

The M_playMenu() method will be tested by check if all the setVisible variable for each of the button, listview, dropdown is true, when playMenu button is set to true.

The M_quitGame() method will be test by simply checking to if the game is closed.

The M_openOption() method will be tested the same way as M_playMenu(), checking if the setVisible of import button and list of robot team is true.

The S_pauseGame() method will be tested by simply checking to see if the game is paused during spectator mode when the button pressed is equal to true.

S_fastForwardGame() method will be tested by checking to see if the game is being fast forwarded on the button input being true.

S_fogofWar_Switch() method will be tested by checking to see of the fog of war has been activated or deactivated on button input equaling true or false.

P_startGame() method will be tested by checking to see if the game activates the start sequence after button input is beginButton input is equal to true.

P_selectedBoard() method

P_selectedColor()

P_setTeamName()

O_ImportReadFileScreen()

O_removeTeam()

G_movefireToggle

G_endPlay()

G_turnLeft()

G_turnRight()

G_forfeit()

GS_exitGame()

G_Fire()

G_Move()

G_cycleLeft()

G_cycleRight()

E_exitEndGameInfo()

Changes

Summary

Testing will be an integral part of our system, ensuring that functionality and interaction between all components are correct. We have identified three major components that we consider to be “hotspots”: the board, the controller, and the interpreter. These are the pieces subject to the most interaction and/or manipulation in the system, and as such, will be tested heavily through interfaces in comparison to other, smaller components. Testing the board will revolve around ensuring that hexes and their contents are correctly modified when needed, as well as keeping

track of general game flow like the current robot playing. Bugs in the board could result in a visual misrepresentation of the game, incorrect actions being performed, and more. The controller will be handling the game's thinking, meaning it is imperative there are no errors occurring within. These tests will involve emulating various user-inputs and properly handling their inputs, as well as properly influencing the model when required. Any errors in this could result in misinterpreting of user-input, incorrect actions being performed, or a wide range of other hiccups in our system. Finally, we will be constructing an interface for testing our robot interpreter. The interpreter will be handling all interactions between our system itself, and the robot teams for our game.