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

Interpreter:

Board:

Desc:

Our testing plan will require an interface to effectivly test our board model class. This interface will perform tests on the board by creating “fake” Hexs and Robots. The Interface will

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, 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, then 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.

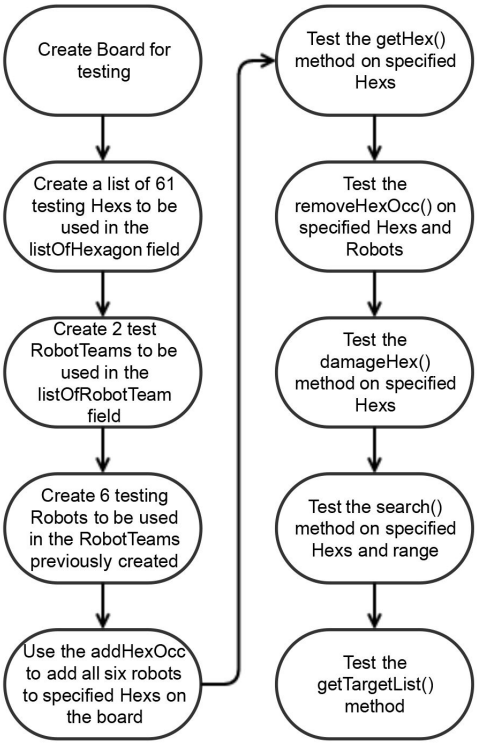
damageHex(): The damageHex() method will be tested by first damaging the previously defined hex with no robots in it. To find the damage stat of the current robot a robotTeam will have to be created and placed into the listOfRobotTeams. This robot team can be defined with a previously defined test robot 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 add 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.

Mock Scenario:



Controller Class:

Robot Interpreter Interface:

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 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 testing robot-manipulating words. These words will involve some change in the robot's state, as well as a possible change in the stack. For testing of these 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. Tests similar to this will need to be performed for all system-defined words to verify correct functionality, as well as asserting any invalid operations are not performed.

Significance
Testing Plan

Things to test for:
- Word list is properly constructed
- Scripts are correctly parsed
    - Ensure that user-defined words are in variable list
- Robot can be manipulated once play() has been called
- Each word in the word list performs its function properly
- The robot performs the correct actions when told to

Controller Interface:
Description
Significance
Test Plan

The controller is the only way for the, so when testing method between the view and the model, we have to create new testing variable for both the view and the model.

The currentTurn() method would be tested by physical 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.

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.

M_quitGame() method will work?

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.

S_pauseGame()