Testing Document

CMPT 370

Group C4

Jack Huang

Brandon Jamieson

Ixabat Lahiji

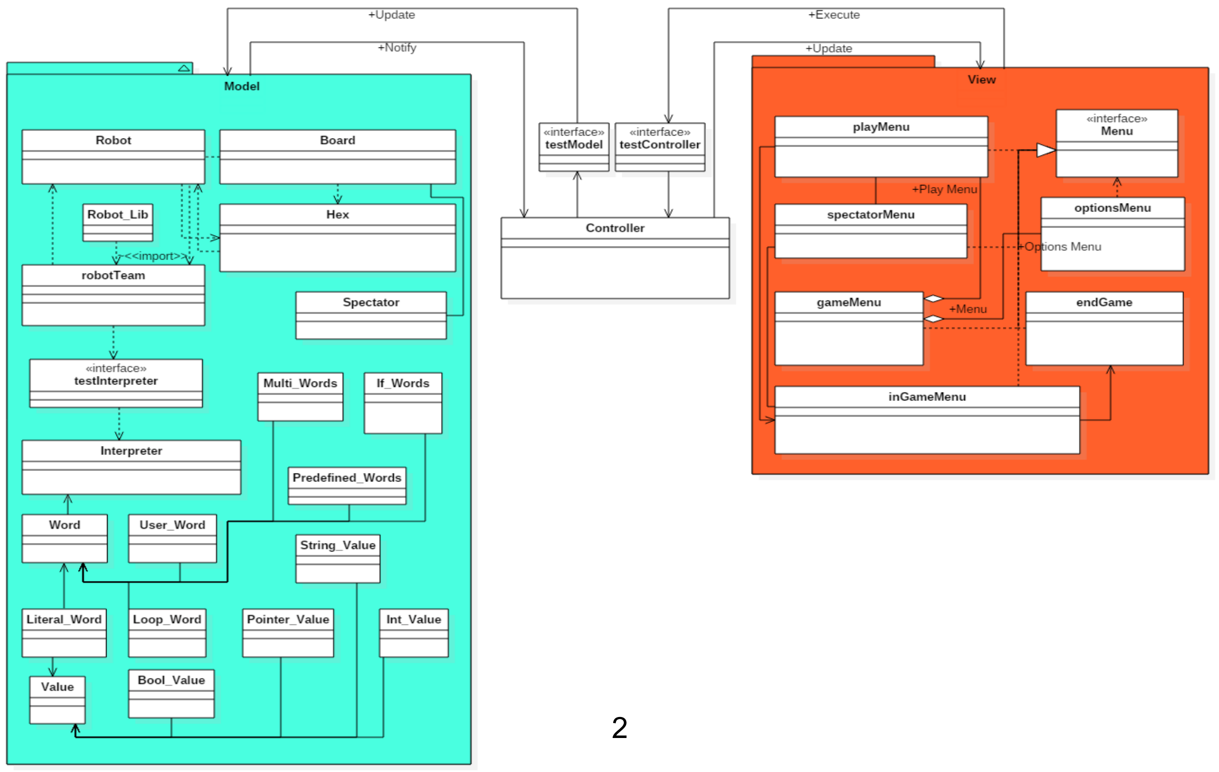
Daniel Morris

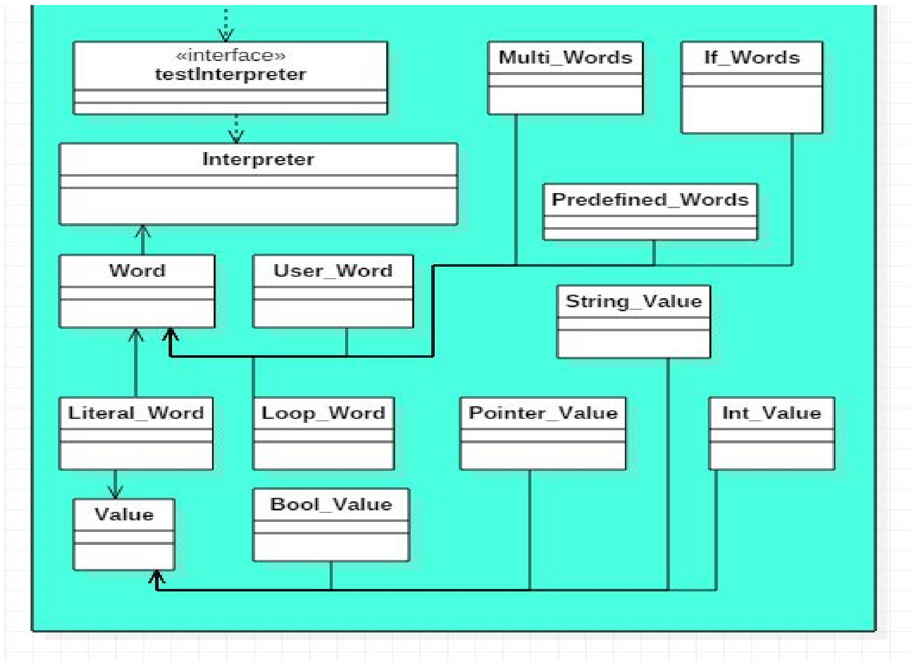
Kevin Noonan

Table of Contents:

Introduction

Testing will serve a crucial role in ensuring our system operates properly. We have identified three components that will need specialized testing. Firstly, we have implemented an interface for our interpreter to test it's functionality along with all the words and values in our system. Secondly, an interface for testing our board at the entry point of the model will be added, simulating controller input and ensuring data is handled correctly. Finally, an interface for ensuring our controller works correctly will be implemented, which will emulate user input for proper responses. We have chosen three components because they will be handling much of the system's data flow and will be more error-prone than others.





Robot Interpreter

Our testing plan will require a specialized interface for performing tests on our 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 robot object will be manipulated by the words executed so that we can check their effect on a robot without the actual robots. This way, we can execute words and verify their correctness by checking the state of the robot object, the stack, or both in some cases. This will ensure that all classes of words work, and that the interpreter is operating correctly.

Unit Testing:

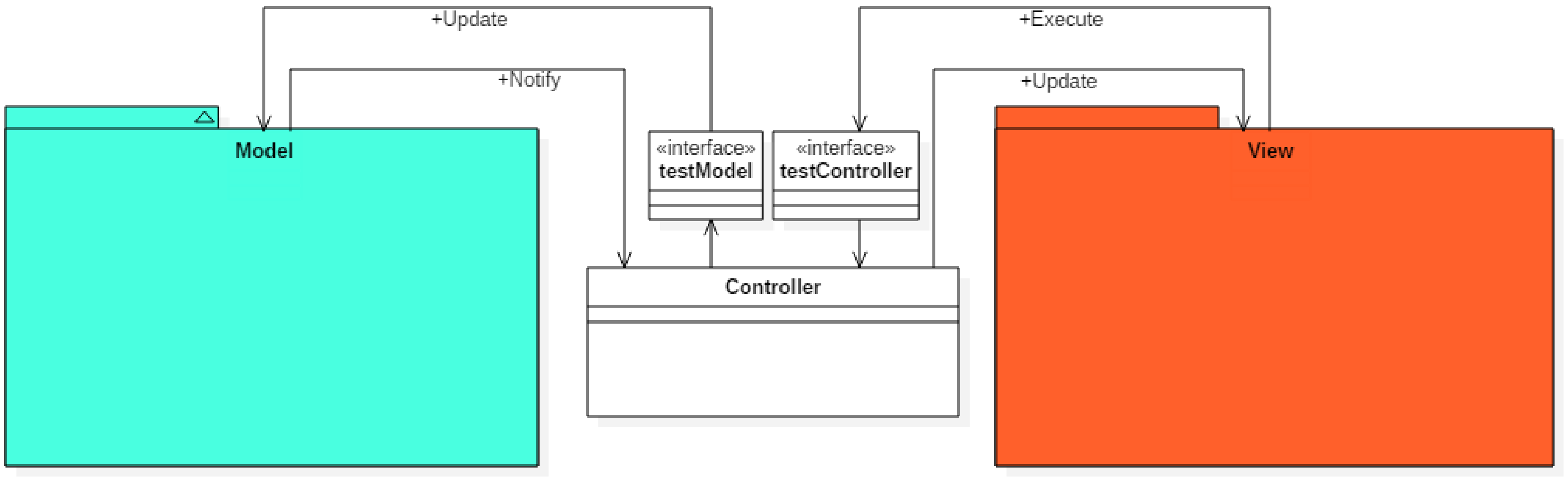
parseScript(): As each robot will have their behavior script parsed at the beginning of the game, we will begin our testing by parsing a basic behavior script containing a range of user-defined words to test along with our system-defined words. To verify that the parse worked, we can simply check that 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.

Words, values, and play(): 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. This way, we can ensure that each different class of word still functions correctly, and that values are interacting with the stack properly.

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 along with 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, it is not possible for our robots to move out of bounds in our game. Therefore, 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 the invalid operation.



Board

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 to define "fake" robot objects, as well as "fake" hex objects. The robot objects will be necessary for moving around the board and ensuring invalid operations are handled. The hex objects 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 plan:

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.

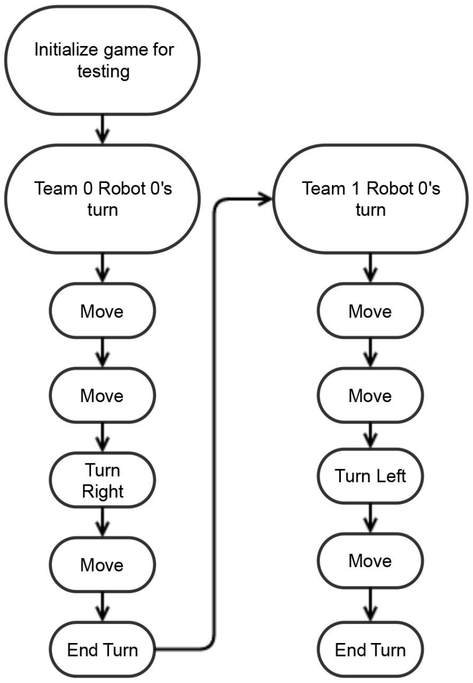
damageHex(): The damageHex() method will be tested by first damaging the previously defined hex with no robots in it. To find the damage stat of the current robot a robotTeam will have to be created and placed into the listOfRobotTeams. This robot team can be defined with a previously defined test robot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

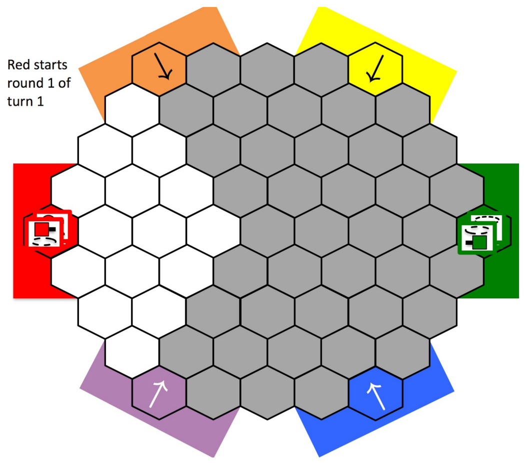
nextRobot(): The nextRobot() method will change the currentRobot and currentTeam numerous times, calling nextRobot() each time and checking that the currentRobot is being changed accordingly.

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

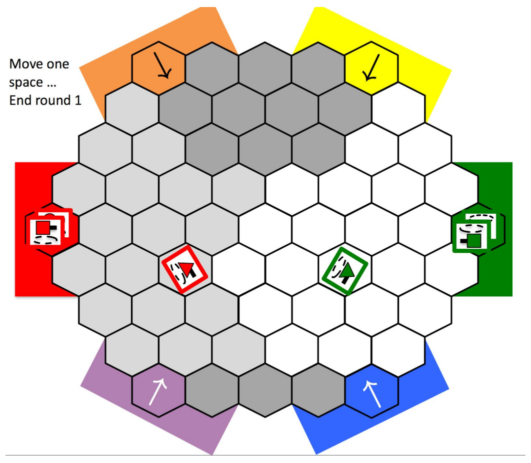
Moving Scenario:



Map of the board at the start of the scenario:

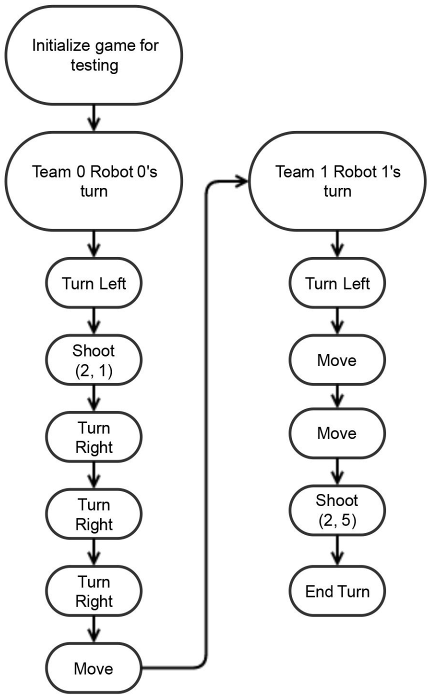


Map of the board at the end of the scenario:

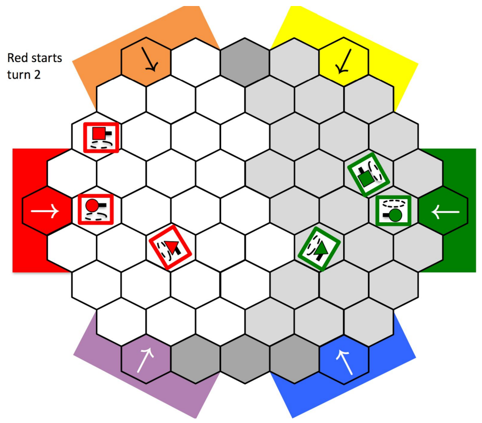


To ensure that the board class is fully capable of handling multiple in a game like scenario we have devised this scenario to do so. The scenario starts with a two team game starting with team zero, the red team in this case. The red scout moves two spaces forward, turns to the right, then moves forward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

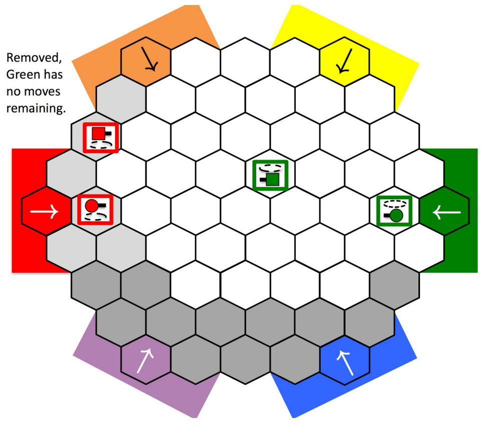
Shooting Scenario:



Map of the board at the start of the scenario:



Map of the board at the end of the scenario:



The shooting scenario allows for an in depth view on how the system can handle the play of robots when they are attacking and killing each other. In this scenario a single round of the game is played and each player loses a robot from their team. This scenario will take advantage of every method inside of the Board class making this a complete mock test of our board class.

Controller

With testing of the model done, our system's controller will need thorough testing. The controller handles all the thinking in our architecture, manipulating the model and the view based on game-flow and user input. As such, to test this component we will need to construct an interface to simulate user input and ensure that the model is properly manipulated. This way we can ensure that user input is handled correctly, and that the rest of the system is appropriately updated in response.

The controller is the most important part of our system. Not only is it necessary for information to be transferred between the model and the view, it also handles all game flow operations such as switching turns, deciding a winner, and much more. This much functionality leads to the controller being a likely source of bugs in our system. Any bugs in this could result in misinterpreting of user input, incorrect actions being performed, both the model and view being updated incorrectly, and much more. This means testing should be in-depth to ensure that our system's gameplay is properly functioning and errors do not occur.

Testing Plan:

Since the controller is the only way for the view and the model to interact, we need to create a "fake" user object that mimic what a 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 methods 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 methods S_pauseGame() and S_fastForwardGame() we would check the difference in the time variable and for the S_fogofWar method we would check if the return value from getFogofWar() method is true or false inside of the Spectator class.

When testing methods 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 checking 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 create an example robot team .json file, to test O_ImportReadFileScreen() and O_removeTeam() making sure the game is able to add and remove teams. For the in game methods, we mimic the user input for each of the button, and in each of the button there will be an test variable that is set to true when it have been pressed. We would first place a robot in the middle of the board and run the tests that would use every in game button, after we would just check the list of test variable for each of the button to see if it had been set to true. The name of the test variable would similar to the button name, so it would be easy tell which button didn't pass the test. We also know ahead of the time where the robot will end up, so we would compare the start location and the end location as our last check to see if all the movement worked.

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 a 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 tested 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 checking the time variable and if it has stopped then we know that the game has been paused.

The S_fastForwardGame() method will be tested by checking the time variable and comparing it. If the time variable has been multiplied by 2x or 4x then we know that the fast forward worked.

The S_fogofWar_Switch() method will be tested by checking to see if the getFogofWar method has been changed to true or false within the spectator class.

The P_startGame() method will be tested by checking to see if the game activates the start sequence also if the variable setVisible is set to true for the InGame_Menu test variables.

The P_selectedBoard() method will be tested by creating separate numbers for the player amount and making sure that the method selects the correct currently selected board size based off of the inputted amount.

The P_selectedColor() method will be tested by checking the test variable within the Robot_Team class and checking to see if it has changed to an integer value.

The P_setTeamName() method will be tested by checking the test string variable in the Robot class and see if holds a string.

The O_ImportReadFileScreen() method will be tested by taking the test .json robot team and attempting to import it to see if it’s successful.

The O_removeTeam() method will be tested by attempting to remove the test .json robot team and then checking to see if it is available anymore.

The G_movefireToggle() method will be tested by using the setVisible variable and using to test variables to see when the method switches between moving and firing mode and then checking true or false on each toggle.

The G_endPlay() method will be tested by using two test robots with int variables and when the method endPlay is used we check to see that the robot int has changed to the next robot's integer.

The G_turnLeft() method will be tested using a test robot on the game board and calling the absDirection two times, once before we use the G_turnleft method and once after we have called the method to check if the integer value is -1 the previous value unless the previous value was 0 in which case is should be 5.

The G_turnRight() method will be tested the same way as G_turnLeft method with the only difference is we will be expecting a positive increment to the integer value unless the value is 5 in which case we will expect 0 as the return value.

The G_forfeit() method will be tested by using test robots that are placed on the board. When G_forfeit is called it will call removeOcc(test robot) and remove the current robot from the current players team. After that the method will call the isAlive method on the rest of team if there are any alive robots left and also remove them from the board until the player has no robots left. If the player was the only user we will use the setVisible variable to check to see if the game board updated from the regular view to spectator view.

The GS_exitGame() method will be tested through the ingame view and the spectator view. We will test this method by using the setVisible variable to use a test value in the main menu. If the value returns true after calling the method in both menus then we know that game has been exited.

The G_Fire() method will be tested by creating an instance of the game board, instructing a robot to shoot a hex and checking that the hex has indeed been damaged, and check our UI booleans to assert that dead robots are no longer displayed. We can then fire again to ensure that the robot does not fire multiple times per turn.

The G_Move() method will also be tested by creating an instance of the game board, except this time we will instruct the robot to move. We can check the destination hex to ensure that the robot has moved, and check that the UI element has moved as well. We can also check that the robot's movement points have been updated, and that the next robot's turn is triggered if we are out of points.

The G_cycleLeft() method will be tested by using a test robot that is placed on the board and another two other test robot's that are placed as an enemy on the board. First the method will run its function and check for move mode or shoot mode. If in move mode we will check the robot's current rotation integer to make sure that is has cycled counter clockwise. If in shoot mode we will check that the currentTarget cycles between the two test robots.

The G_cycleRight() method will be tested similarly to G_cycleLeft method the difference is we will check to make sure the current rotation is cycles clockwise with the test robot and that it goes to the next enemy robot.

The E_exitEndGameInfo() method will be tested by using the setVisible variable to check that when the method is called that we brought to the Game_Menu and that the setVisible is set to true.

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. Firstly, 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. This means it will be subject to constant use during gameplay, errors in which could lead to robots not performing correctly.

Secondly, we will implement an interface for testing the board. This will revolve around ensuring that hexes and their contents are correctly modified when needed, as well as keeping track of general game flow such as the current robot playing. Bugs in the board could result in a visual misrepresentation of the game, incorrect actions being performed, and other problems. Finally, we will be implementing a third interface to test our controller. 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 ensuring they are properly handled, as well as correctly 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.