Professor Christopher Dutchyn

Tutorial Leader Jonathan Baxter

CMPT 370, University of Saskatchewan

November 6th, 2016

ROBOSPORT370: TESTING PLAN

Group B1: Adam Ronellenfitsch, Dylan Prefontaine,
Evan Snook, Matthew Frisky, and Wynston Ramsay

Table of Contents

Introduction

Following up from our Design Document we would like to make sure that implementing our project goes smoothly. For this to happen we highlighted important interactions in our system so that we can test them and understand exactly what we need from them. Doing so will reduce the amount of overhead the group will face when writing actual code for RoboSport370. As a group, we decided to test: Board, ForthInterpreter, GameMaster, HexNodeTraversal, Librarian, RobotAI, and RoboBuilder. These classes are involved in the most common and most important interactions in our design; creating a safeguard for them will make implementation less tedious and more bug proof.

Board

Description

The Board's main purpose is to represent a model of an abstract data type that is made up of HexNodes. In our MVC architecture, the board is a model so it is only used to store data. A board is used to hold a series of HexNodes to form a graph to represent the game board. There are two size options for the length of each side that are determined by the enumerator BoardSize; one size will be a length of 5 hexagon tiles and the other, a length of 7. All the tiles will be linked together with the known size to form a Board model.

Significance

linkNode

The linkNode method is used when initializing the Board to link together HexNodes. As arguments, it receives two HexNodes, "cur" and "other", as well as an integer called side. "Cur" represents the current node that is being linked with the "other" HexNode. The Board constructor repeatedly links nodes together until the board is complete. This method needs to be tested because we want to make sure that none of our nodes on the board end up being null or linked to the wrong spot because that would cause our constructor to fail.

Board

This is the constructor for the Board, it takes a BoardSize as input take is determined by the user. The constructor will then create a root HexNode. It uses this reference to iteratively create HexNodes that will expand from the root until the size condition is satisfied. Once the Board is created, it identifies and stores the corner HexNodes into an array of six. We are testing this method because if the board is not properly created and there are gaps or null spaces in the board, the game will not function properly if at all.

Testing Plan

linkNode

Test 1: cur node is null - create testNode1 as a null node and assign a non-null node testNode2 to its side 0. assert that there is an exception thrown that cur node cannot be null. We need an exception here because we can end up with errors when we try to assign things to a null reference and we need to catch and deal with that situation

Test 2: other node is null - create testNode1 and assign a null node, testNode2, to the side 0 of it. assert that curnode.r == 0. This will ensure that the function does not throw an exception when other node is null and properly assigns it.

Test 3: link side 0 (r) - create testNode1 and testNode2 and assign testNode2 to side 0 of testNode1. assert that testNode1.r == testNode2. We must test this because it is a lower boundary case for our integer side.

Test 4: link side 3 twice (r) - create testNode1, testNode2, testNode3 and assign testNode2 to side 3 of testNode1 then reassign testNode3 to side 0 of testNode1. assert that testNode1. r == testNode2. We must test this because our board is set once it is created and we should not have to override nodes as that could result in some shenanigans.

Test 5: link side 5 (ur) - create testNode1 and testNode2 and assign testNode2 to side 5 of testNode1. assert that testNode1.r == testNode2. We must test this because it is an upper boundary case for our integer side.

Build

Test 1: size is not 5 or 7 - test calling a constructor on testBoard with a size other than 5 or 7. assert that and exception was thrown and that testBoard == null. we need to do this because there is no reason for us to create a board with anything other than a size of 5 or 7.

Test 2: size is 5 - test creating a testBoard of size 5 and assert that you can travel 4 nodes away from the root node in any direction before you hit null.

Test 3: size is 7 - test creating a testBoard of size 5 and assert that you can travel 4 nodes away from the root node in any direction before you hit null.

ForthInterpreter

Description

The ForthInterpreter will be taking input from Robots as Strings and executing the commands on its Stack using its Words while gathering its information from the GameMaster.

Significance

execute(String forthString, HashMap<String, String> userDefinedWord)

Execute a forth string that contains multiple words which get passed to executeSingle to be executed individually. This forthString may contain a user defined word which will be extended out before being sent to executeSingle.

executeSingle(String singleCommand)

Execute a single forth command. We are testing this so that we can be sure that it works for all kinds of words.

Testing Plan

execute

Test 1: null string - assert that an exception is thrown. This is necessary so that we are not executing scripts with undefined words.

Test 2: Forth String - assert that the command is executed. This test ensures that the functionality of pre-defined words.

Test 3: Defining Word - assert that the command is executed. This test ensures that the functionality of Defining Words works.

Test 4: Using defined words - assert that the command is executed. This test ensures that the functionality of user defined.

Test 5: Containing comments - assert that the command is executed. This test ensures that the user can write comments in a script and still have it work

executeSingle

Test 1: null string - assert that an exception is thrown. This is necessary so that we are not executing scripts with undefined words.

Test 2: Literal Word - assert that the command is executed. This test ensures that the functionality of Literal Words works.

Test 3: Defining Word - assert that the command is executed. This test ensures that the functionality of Defining Words works.

Test 4: Forth Word - assert that the command is executed. This test ensures that the functionality of Forth Words works.

Test 5: Comments - assert that the command is executed. This test ensures that the user can write comments in a script and still have it work.

GameMaster

Description

GameMaster is the controller of the overall game mechanics such as: controlling robots, (turning, shooting, firing, scanning and identifying) and communicates with the ForthInterpreter to execute commands for the RobotAI.

Significance

Scan

Scan returns the number of robots currently in view of the current robot. This function is crucial for AI robots to know their surroundings. Therefor it is necessary to test this function thoroughly.

identify

Identify will take in an index for which robot to pick. Then it will push three things: its TeamColour, how far the robot is away and the direction the robot is in, as well as the robots remaining health. Identify, like Scan must be tested thoroughly. We need this method tested so that we can be sure that our AI can appropriately see enemies and ally's on the map.

Testing Plan

Identify

Zero robots in range:

Test 1: identify with index = -1. - nothing should be added to the stack.

Test 2: identify with index = 0. - nothing should be added to the stack.

One robot in range:

Test 3: identify with index = 0. the only robot's stats should be added to the stack.

Test 4: identify with index = 1. - nothing should be added to the stack, accessing past the number of robots scanned.

Three robots in range:

Test 5: identify with index = 0. the first robot's stats should be added to the stack.

Test 6: identify with index = 1. the second robot's stats should be added to the stack.

Test 7: identify with index = 2. the third robot's stats should be added to the stack.

Scan

Test 1: scan with no robots in range. - create an integer testInt and assign it the result of the scan it should be 0.

Test 2: scan with one robot in range. - create an integer testInt and assign it the result of the scan it should be 1.

Test 3: scan with three robots in range. - create an integer testInt and assign it the result of the scan it should be 3.

HexNodeTraversal

Description

HexNodeTraversal is used to iterate through the graph-like structure of the HexNodes. It keeps track of the root HexNode (where the traverse began), the current HexNode that is being traversed, and which linked node it is facing (see Global Direction). It provides movement through the Hexagon board by moving from HexNode to HexNode, layer by layer while preventing the traverse from leaving the boundaries. It begins from the center and starts from the global position of 0 and works around that layer, then moves to the next layer until the entire graph has been traversed. It is an inside-out approach at traversing the Board. First layer will just be the middle so it will have the node 0. The

second layer will have 0-5, the third layer will have 0-11, the fourth 0-17, the fifth 0-23, and in the case the board size is 7 the sixth layer will have 0-35.

Significance

HexNodeTraversal

This is the constructor for the HexNodeTraversal class. It simply creates a HexNodeTraversal object, initially sets the root HexNode and the direction to start the traversal. We will be testing this method so that we know whether the Board was properly traversed and each node was touched.

go

Go is a method that traverses to a HexNode. Distance out being which layer it is on and distance around being the index of the node on that layer. Testing this method to ensure that we are reaching the proper layers and nodes.

Testing Plan

HexNodeTraversal

Test 1: root is null - assert that an exception is thrown. This test is just in case something goes wrong, we want the game to catch that it has a null value.

Test 2: root is the only node that exists and direction = 0 - assert that traversal returns the root. This is necessary because it is our boundary case in which the traversal will exit from.

Test 3: root has one element on the next layer – the traversal returns the element in the next layer. The test ensures that we can navigate out at least one layer.

Test 4: root has full next layer - assert that the traversal returns the last element in the second layer. The test ensures that we can navigate a full next layer.

Test 5: root has an item on the second layer - assert that the traversal returns the element on the last layer. the test ensures that we can navigate out many layers.

go

Test 1: root node is null - assert that an exception is thrown. If no exception is thrown, then we can call the function on a null node which can lead to many bugs.

Test 2: distanceOut is greater than boardsize 1 - assert that an exception is thrown. If no exception is thrown, then we can call the function on a null node which can lead to many bugs.

Test 3: distanceAround is greater than distanceout * 6 - assert that an exception is thrown. If no exception is thrown, then we can call the function on a null node which can lead to many bugs.

Test 4: distanceOut is 1, distanceAround is 0 - assert that the node returned is the first node in the next layer. this ensures that go can go out a layer and that it can find the first node in a layer.

Test 5: distanceOut is 3, distanceAround is 18 - assert that the node returned is the 18th node in the 3rd layer. this ensures that go can go out many layers and that it can find the last node in a layer.

Librarian

Description

The Librarian has an array of all the robots in the system and has procedures to add, remove, update, and load in the robots to the list. It can also create a JSON file for the robots to have their information saved.

Significance

addLocal

This method adds a robot to the list of robots. We are testing this because we want to ensure that robots being added are viable and won't break the game.

removeLocal

This method removes a robot from the list of robots. We are testing this method so that we can be sure that the correct robot is being deleted because we do not want to accidentally delete information.

updateLocalRobots

This method updates the robots saved on the local machine. We are testing this method so that we can be sure that the correct robot is being updated because we do not want to accidentally delete information.

toJSONArray

This method turns all the robot's information into a JSON formatted file. This needs to be tested so that we can ensure that the proper information is being transferred.

Testing Plan

addLocal

Test 1: Robot is null - nothing should be added to the list, an exception should be thrown.

Test 2: Robot is identical to a robot already in the list - nothing should be added to the list.

Test 3: List is empty - a new list should be created containing the robot.

Test 4: List exists with robots already added - the robot should be added to the end of the list.

removeLocal

Test 1: List is empty - nothing should happen to the list.

Test 2: List has 1 robot - robot should be removed.

Test 3: Robot is at the end of the list - robot should be removed.

Test 4: Robot is at the beginning of the list - robot should be removed, the remaining robots should be moved.

Test 5: Robot is in the middle of the list - robot should be removed, the remaining robots should be moved to account for this.

Test 6: Robot is null - nothing should happen, an exception should be thrown.

Test 7: Robot does not exist in the list - nothing should happen.

updateLocalRobots

Test 1: No robots saved to the local machine - nothing should be changed.

Test 2: One robot saved to the local machine - one robot should be updated.

Test 3: Multiple robots saved to the local machine - multiple robots should be updated.

Test 4: Multiple robots saved to the local machine but only one has changed - only one robot should be updated.

toJSONArray

Test 1: Robot missing information - an exception should be thrown.

Test 2: Fully functioning robot - JSON formatted file should be created.

Test 3: Robot that already has a JSON formatted file - file should be overwritten.

RobotAI

Description

RobotAI is an extension of the Robot class that contains extra information about this robot including the Team that coded it, its previous stats and the script that created it.

Significance

toJSON

Get a JSONObject representation of this Robot. Testing this ensures that we have properly created a robot and exported it to JSON file which can in turn be read and interpreted as a robot.

Testing Plan

testRobotAI = RobotAI("SomeJSON")

Test 1: someJSON is a valid script - The InitCommand and the play words must be defined. All the defined words in the script must be in the words variable. A mailbox must be defined.

Test 2: someJSON is an invalid script - making sure we catch an invalid script using a thrown exception.

RobotBuilder

Description

RobotBuilder is part of the Model in our architecture, it gives us an easy interface for creating robots and modifying single aspects about them without needing a gargantuan constructor for our Robot class.

Significance

Robot build

This method returns the current Robot with the new changes made to the robot's variables. We assume that the RobotBuilder constructor calls the right function depending what on the boolean value of "isAI" passed as a parameter. For Robot build to be called it, false needs to be passed as a parameter. If this method is called it will create a Robot object, so we must make sure that it meets the needs of a regular Robot since we are using a builder class. Testing this method ensures that a robot is built with no null values

RobotAI build

This method returns the RobotAI with the new changes made to the robot's variables. We assume that the RobotBuilder constructor calls the right function depending what on the boolean value of "isAI" passed as a parameter. For RobotAI build to be called it, true needs to be passed as a parameter. Since we are using a builder to create a RobotAI object, we need to assure that a "normal" RobotAI is created. Essentially making sure that the object instantiates a complete RobotAI class. Testing this method ensures that a robot is built with no null values

Testing Plan

Robot build

Test 1: Build a robot with all null variables - assert that an exception is thrown for each variable. This ensures that we do not end up with a robot that has any null values.

RobotAI build

Test 1: Build a robot AI with all null variables - assert that an exception is thrown for each variable. This ensures that we do not end up with a robot that has any null values.

Changes

After the Design Document and the feedback our group received, we did not feel the need to make any major changes to our system. The only small change we have made since then is that HexNodeTraversal is now an iterator. Also, we did not include AspectJ in our design because we are not familiar with it and were unsure about its usage. In our implementation, we may decide to use AspectJ, but there is no extra testing to be done for it; the functionality of our system will remain the same whether we decide to use this tool or not.

Summary

In our testing for RoboSport370 it is important to point out we did not include trivial tests. As stated in the introduction, only the important and common interactions were tested. We felt that it was sufficient to have a plan for testing these interactions only because when we go to implement it, writing code for tests will be easy; the trivial stuff is self-explanatory and now that we have an idea of how to test the hard stuff, testing our software will be much easier. Just to reiterate, the classes we developed a test plan for are: Board, ForthInterpreter, GameMaster, HexNodeTraversal, Librarian, and RobotAI. With the completion of the test plan, next we can start our actual implementation of RoboSport370; writing code for this will feel "safer" and easy now that we have a safeguard to protect our design.