CMPT 370 Testing Plan Document for BattleBots

Group A5
Mackenzie Power
Will Revell
Haotian Ma
Ryan Tetland
Mitchel Kovacs

Phase3_TESTING

Table of Contents

Introduction …………………………………………………………...3
View Testing…………………………………………………………..3
Controller Testing……………………………………………………..3
Model Testing……………………………………………….........…4

- GameBoard and Hex classes ……………………….....……….4
- Robot and Gang classes……………………………….……….5
- RobotStats and StatTracker classes………………………………6

Changes Made…………………………………………...…………7
Summary ………………………………………….....…...……7

Introduction

This document will outline the methods and actions we will take in order to successfully test the various functions involved our BattleBots program. To test our game BattleBots we will have to approach each section in the Model-View-Controller differently. Since we are using Test-Driven-Development we will be separately designing tests for our Model, View and Controller which we will implement before any of the game code. This means we will be able to minimize the amount of code required for the game to function correctly. Due to the nature of our design, certain aspects of the game will not be testable until other classes that they rely on have been created. For certain classes we can avoid this by generating dummy classes, which are simplified classes used only for testing, which will be implemented if the required class has not yet been finished. In other cases, we will produce mock classes, which are classes where variables are set to permanent values, to allow easier testing of specific classes.

View Testing

The View is the outermost interface that interacts with the user. It displays the GUI's for the game and interacts with the user by taking in input and sending that information to the Controller.

The class TeamSelectionPanel, which is part of the View, also interacts with the Robot Librarian. The user will have the options of uploading and downloading robots from the Robot Librarian, as well as creating new robots, and editing existing ones. The TeamSelectionPanel is significant because the game cannot be played without being able to use robots from the Robot Librarian, and this is why it must be properly tested. The functionality will be tested in the Controller interface, but we will have to check visually that the proper information displayed in the graphical user interface.

The GamePanel class is also part of the View. It is responsible for drawing the state of the GameBoard class as well as the statistics on the side. It will redraw the GameBoard when the updateGame function is called. This is a significant class for when the game involves human players so they can see an accurate state of the GameBoard. We will visually check if the GamePanel displays the correct state of the board.

Since we do not have the resources to properly test that these GUI's properly display exactly as intended, we can only visually check them with our own eyes. When buttons are clicked, the Controller will test the input and we will check if the GUI's behave properly. This is how most of the View interface will be tested.

Controller Testing

The Controller contains a grouping of classes that rely heavily on classes in the Model and View to have already been finished before they can be properly tested. Therefore several "dummy" classes will have to be created to allow for testing certain aspects. Also due to the Controllers nature as an interface between the Model and View certain functionality will only be testable after the implementation of most of the Model and Views classes. The Controller directly implements three interfaces: KeyListener, ActionListener and GameObserver. KeyListener simply needs to be able to perform an action if a key on the keyboard is pressed. Therefore it can be initially tested to see if it correctly determines that a specific key has been pressed and perform a generic action to confirm it being pressed by printing a string with the key name. After the robot class has been implemented the move function will then become testable, upon key press we will check that the robot has moved in the correct direction by checking the robots new verticalLocation and horizontalLocation values and comparing them to the before movement verticalLocation and horizontalLocation values, as well as performing a visual test on the GameBoard. The ActionListener's job is to determine if a mouse click has taken place and what location has been clicked, whether it is a Hex on the game board or a navigation button. To correctly test this, a "dummy" GUI panel will need to be created to test whether the action is being correctly interpreted. This will then be checked visually. Unfortunately, actions during the actual game will not be testable until after the Jpanel, GameBoard, Hex, and Robot models have been implemented. Once they have been it will be tested whether when a Hex with one or more robots is clicked on the health of each robot is correctly lowered. The GameObserver only checks whether or not the game has ended by checking the number of teams with robots that are still alive, it returns a boolean value of true if there is only one team left and the game has ended or false otherwise. To test this we will need to create several "dummy" team classes that only contain an integer value for the current number of alive robots and see if GameObserver returns the correct boolean value with each scenario. After the View and models have been completely finished we will be able to visually test if the game ends when it should. For the GameController, each of its handleEvent functions and the gameOver function will be tested through their corresponding interfaces, whilst the startGame   function will not be testable until the View and the MainMenu Jpanel have been completely implemented, once they have been we will visually test that the correct start screen has popped up.

For the team selection, robot AI and the Robot librarian to work properly the Game Controller must check whether or not there is a JSON file present, and whether or not it is empty. For the testing we will check if there is a JSON file in the corresponding directory, if there is not we will throw an exception. There will then be a test to determine if there is any text in the file throwing an exception if there isn't.

Model Testing

The Model in our architecture consists of several object classes that interact with each other through the Controller in various ways. The conglomerate of classes that comprise the Model are: GameBoard, Hex, Robot, Scout, Sniper, Tank, ScoutAI, SniperAI, TankAI, Gang, and GangAI, Many of them have attributes of other classes within the Model. Many of these classes can be tested using J-Unit tests to ensure correct functionality. Since our main Integrated Development Environment (IDE) used to create the program is NetBeans, it has a function that is able to generate setters, getters, and J-Unit tests for each function that is used in a given class. Due to time constraints, we will not create J-Unit tests for any getter and setter functions. Since the IDE is able to automatically generate those functions, we can assume safely that they should be able to work. Testing for these functions may be considered if we become ahead of schedule during implementation. However, these functions will be important for the testing of other functions that will result in changes of attributes of other classes.

GameBoard and Hex class:

The simplest functions to test for in the game will be those that are associated with the GameBoard class and Hex class. As discussed, simple J-Unit tests can be used to determine if these classes are able to be constructed correctly. With the GameBoard class, we will have to make sure the J-Unit tests check to see if the board is able to instantiate with the correct size and number of Hex spaces (when given a specific integer). This will be done by asserting within the test that all playable spaces exist (are not equal to null), and spaces that are not used will be set to null (since our game board is a 2D array of Hex spaces). For the Hex spaces, it has only the one function which is to check to see if the Hex List of robots it contains is empty. Since the function simply returns one of three different String values, the J-Unit test will assert a specific string depending on if there is a robot present or if the Hex is out of bounds in the game. For this test and many others, we will have to create mock objects of the Robot class to test the functions that involve use of robots. Within the J-Unit test, we will have to instantiate a new robot and then hard code it into the Hex object list and check that if can return a correct value.

Robot and Gang classes:

There will technically be no objects in our program that will be of the Robot class. There will, however, be many objects of classes that inherit from the Robot class. All of those classes will continue to utilize most of the functions from the Robot class. As mentioned previously, most of the functions involved in these classes are setter and getter functions and will therefore not be tested individually. We will, however, perform a J-Unit test for the constructor of each of the following classes: Scout, ScoutAI, Tank, TankAI, Sniper, and SniperAI. Since we already know what the various attributes of each robot should be (range, health, damage) we will construct each Robot class and determine whether or not each constructor can successfully set the various parameters correctly, since each child class has a new constructor that is redefined. These J-Unit tests will use the assertEquals functions in Netbeans to check that each and every attribute has been accurately assigned based on what we know the attributes should be.

The receive damage function should be able to take away a specified amount of damage from robots that call the function. J-Unit tests for that function will make sure that the appropriate amount of health is subtracted from each robot and test that robot health cannot go below zero. During turn moves and scans, alive robots that are in play will be determined by having health that is greater than zero. An exception should be caught whenever the function is called on a robot with no heath since we need to consider the possibility of a Hex space being attacked that contains both an alive robot and a dead one.

The AI robots (ScoutAI, TankAI, and SniperAI) all have the additional functions of move, turn, scan, and shoot. Since these functions require the use of mock robot classes to test their ability to function correctly, we will have to create mock game simulation to ensure that those functions work appropriately.

Since the GameBoard class will always be created and used in every game and simulation, we will create all tests that involve interaction between Model classes here as a testing main function. Note: for all exceptions that will be thrown for these series of tests, each new exception thrown due to expected values not matching resulting values will have an accompanying message that prints out a detailed message of what the two compared values were and what test was failed. Robots AIs must be able to move, shoot, scan, and turn. We will have to be realistic with our testing because if we wanted to test every conceivable action and move at every position then we would likely be creating tests for years. Since we have only a month for coding, we will have to only test basic scenarios for each function and the boundary conditions for them as well. To do this, we will have to hard code in a couple of mock robot AIs (two of each type) and assemble them into two different GangAIs. Since each Gang class will have a Scout, a Tank, and a Sniper, to test this class we require a mock robot.

The mock robots will need to have a field that will differentiate it from the others on different teams. This is important, as we need to make sure that there is only one of each robot per team. Since there is only one of each robot, we will test that there is exactly three robots per team.

The GangAI class is very similar to the Gang class. The main difference is that the GangAI class is for AI robots. It is important that a GangAI class consists of only AI robots, so in the mock robots there will need to be a field that indicates if the robot is an artificial intelligence. The regular Gang class will have to make sure that it doesn't have any AI robots. To do this, we will compare the type classes of each robot and throw an exception if there is any discrepancy. Under no circumstances should this exception actually occur as the constructor for the gang classes should prevent this from occurring. If it does occur, then it will indicate that we have a serious bug involved in assigning the correct classes to the Gang class.

From here, we will initialize a new GameBoard and code them on opposite sides of the board. From here, we will be able to test execution of the various functions the robots employ to play the game. We will then execute the move function for each robot, attempting to move them to different parts of the game board, including spaces that go beyond the boundaries of the board. Since moving outside the board will throw an exception that is caught within the move function, we should test using asserts that the position of the robot remains unchanged when attempting to move to these positions. Since every robot retains the previous position it was in, we will have to test that is has changed. In addition, we will have to test that the robot remains in its current position if all of its moves are used up. These tests can be done by throwing exceptions when the value obtained does not match the expected value.

The scan function will return word values based on whether or not there are any robots present in a scanned Hex space. These series of tests will involve moving the robots around to different spots on the board and see if the scan function can correctly return the information. We will have to test that it will return correct values for when there are no robots present; one robot present; two robots present; one single dead robot; and also if there are dead and alive robots present. If the returned values are not equal to the values we expect, then exceptions will be thrown.

Testing the turn function should be a relatively straight forward process. A simple J-Unit test would suffice, but since it is part of every move a robot AI makes we will include it into these series of tests. Since each robot will have an attribute known as the relativeDirection which (as the name implies) contains the robots pointing direction, all we need to do is check that the value is incremented by one each time scan is performed. If it is not, then an exception will be thrown.

The final function which will be needed for this series of tests will be the attack function. We must test that a robot will not fire upon a Hex space that contains either no robot or only dead robots. For this we will have to compare the shotsLeft attribute on the robot to ensure that no shots were allowed to fire. If they were, then an exception will be thrown. Next, for each robot type, we must ensure that they inflict the correct amount of damage. To test this, we will have robots call their attack function on a robot in range and then compare the shotsLeft attribute and the health attribute of the robot being attacked then throw an exception if the values are not what is expected. We will then need to perform this on attacking Hex spaces that have more than one active robot; and also on spaces that have both an active and dead robot on them and compare those values.

<u>RobotStats and StatTracker classes:</u>

The last class which will need to be tested from the Model will be the RobotStats class, which is a list of stats that each individual robot will contain (if it is a robot not controlled by a human). This class will be tested last once all other functions work correctly since it involves all other classes and functions to work before it can accurately record stats. To test for this correctly, we will create a blank stats page for each robot and then hard code in what the expected stats for each robot will be after all the actions done during the simulation test. We will throw exceptions whenever an expected value differs from the actual value. To initially test the RobotStats class we will create a spreadsheet of values for each of the stat integers being tracked; kills, deaths, games played, games ended alive, wins, loses, damage inflicted, damage absorbed and distance travelled. We will check that each value has been correctly incremented from zero and if it has not been then we will throw an exception. After the GameBoard, Robot and Hex models have been completed we will then be able to run the testing scenarios within the main testing function in the GameBoard as talked about earlier. This will allow us to track whether stats values are being correctly updated during the move and shoot actions as well as the game end scenario.

The StatsTracker class contains only an array of every robot stats class in the game. Initially we will only test whether the array is the correct size for a given number of robots. For further testing the RobotStats class must be completed, after which we will test whether each array member holds the correct robot, and if it is NULL then we will throw an exception.

Changes Made

During our test planning, it was discovered that there were several important design decisions made previously that we should have changed or implemented. Most of these functions were relatively minor, and none will affect the overall design of our game. The first few changes we have decided to make were to the Robot class. It was forgotten in our design that each class of robot should have a reference to the game board that it is playing on. Without this reference, it cannot actually interact with the game in any real capacity. The Robot class and all inherited classes will now contain a new function called the receiveDamage function which will take in an integer value representing the amount of damage a robot will receive. This will make it easier to deal damage to robots. In addition, move and attack functions will be added to the robot classes Scout, Sniper, and Tank which will be used alongside action listeners when users use mouse clicks and key inputs to move and attack (respectively). Another change we will have is that both the GangAI class and the Gang class will take in additional values in their constructor. These values will be references to the robots that will be on the same team. Lastly, we have decided that instead of containing an array of Robots, the Hex spaces will instead have a list of Robots (which will be easier to access and remove).

Summary

In conclusion, our testing will be based strongly on Test-Driven-Development. To do this, we have decided to use a combination of J-Unit testing and visual testing to ensure correct functionality. J-Unit testing creates a separate class that involves: setting up the class that will be tested; testing that class; and then tearing everything down. This style of testing is convenient and works well for testing the model. The J-Unit testing will not work with the view, however, so to test the view we will check each panel, and decide if it looks as anticipated. Many of the classes in the model and the controller are dependent on other classes. To test classes that have dependency's, we require either a "mock" or "dummy" class. We utilize dummy classes for operations in the controller, but for aspects of the model that involve significant class interaction we will utilize mock game objects of different classes to simulate various actions and functions executed during the game. While constructing this document, several aspects of our program's design were discovered to be lacking or incorrect. Most of these aspects involved creating new functions and attributes to a few of the classes already established (especially the Robot class). Since we are developing software using Test-Driven-Development, the construction phase will be more streamline and time efficient leading to better designed code.