Robo Sport Testing Plan

Software Engineering Team: D2

Arianne Butler
Kristof Mercier
Michael Graham
Chris Mykota-Reid
Samuel Horovatin

Contents

Introduction

Robot Sport is a computerized board game that is under development for CMPT370 at the University of Saskatchewan. Its purpose is to demonstrate the practicality and importance of the Software Engineering process to student Software Engineering teams. The game consists of a mixture of human and AI players, each controlling their own gang of robots on a hexagonal battlefield. Robot gangs work together to overcome the fog of war, and defeat any and all robots that are on an opposing gang. Robo Sport is primarily a game of strategy, and as such, it should challenge the users who play it.

Objective

The objective of this document is to outline and explain the testing plan of the Robo Sport software system. It will discuss the various test targets of the system, as well as the testing strategy that will be deployed to verify the correctness of the targets. A test target is defined as any aspect of the system that requires some form of testing to verify its correct functionality. The testing strategy will employ multiple methods of testing, where each unique testing method is responsible for verifying the correctness of different areas of the system. Testing will include unit testing, user interface testing, database integrity testing, integration testing and multiplayer functionality testing. With regards to unit testing, this document will provide an English description of the classes which require less testing, and a more comprehensive overview of individual functions for the classes which require more rigorous testing.

Testing Targets:

Classes

- Robot
- Gang
- GameBoard
- StatsLogger
- StartScreen
- SetUpMenu
- GameScreen
- PassTheController
- EndScreen
- DrawRobot
    - DrawScout
    - DrawSniper
    - DrawTank
- GameMaster
- InputController
- SetUpMenuController

- StartScreenController
- EndScreenController
- Interpreter
- InterpreterFunctions

User Interface

- The interface requires light user testing to verify that it is functional, intuitive, and discoverable.

Integration

Each of the systems individual modules will need to be integrated and tested as a whole once all unit testing is complete.

Database

- The system can upload and download various computer AI's and statistics from a server. This functionality will need to be verified.

Multiplayer Functionality

- The system must be tested for smooth multiplayer functionality, with consideration for both human vs human games and human(s) vs AI(s) games.

Testing Strategy:

The testing strategy for our system will include unit testing for each class, user interface testing, integration testing for separate modules, database testing, and multiplayer functionality testing on the various human(s) vs AI(s) combinations. Unit tests will be conducted on the classes within the Model, Controller, and Interpreter components and will use Black Box testing. The View component, or the user interface, will be tested by visual confirmation from members of our team, to ensure that all interface interactions happen as expected. Integration testing will begin from the lowest level or our architecture to the highest level, using a bottom-up approach. Database testing will be handled separately, and as such, will occur in a separate class. This class will upload and download a document from the server, and check to see that the process worked as expected. Multiplayer functionality requires light visual testing. Members from our team will sit down and play the game with the varying combinations of human(s) vs AI(s), verifying that the multiplayer functionality is correctly implemented.

Unit Testing

Robot Class

As the robot class is one of the most instrumental classes in the system, it requires careful testing to ensure that each method returns the correct value. As a result, we have decided to use Black Box testing on the Robot class. We made this choice because a robot is instantiated using a JSON file which can be considered the black boxes' input. As the structure of the JSON is known, we can use it to determine what the output of all of the public functions of the robot should be. For example: getTeam() should return the software team's ID that is stored in the JSON file, and robotID() should be the unique combination of the robot's name, teamID, and gangID.

For testing more abstract functionality like the robot's messaging system, we will have a single robot send multiple messages to itself, and then have it check its own inbox. If the messages are returned in the correct order (ie. the first message sent is the first message received) then the messaging system is running correctly. Similarly, things like the robot's health, range and damage can be tested in the same fashion. The robot can be forced to shoot itself, and upon doing so, the system can be queried to check if the robot's health has been decreased by the correct amount. We can also query the robot to shoot again to confirm that it only one shot per turn is enforced.

Gang Class

Testing the Gang class will be fairly straight forward. Like the robot class, we will check to ensure that the input to the class is the same as the output. As the information is never changed once a Gang is created, this means that it should remain the same.

StatsLogger Class

We will confirm that the Stats Logger properly stores and sends the data it reads from its function calls, since stats logging is a key part of our system requirements. We will use Black Box testing to verify each of the methods, because none of them contain any difficult control logic that would require White Box testing. We will feed the StatsLogger class phony inputs and check to make sure that StatsLogger properly returns a JSON file with the correct values (based on the phony inputs). Overall, the class will need test cases for 6, 9, 12, 15 and 18 robots. This will ensure that the StatsLogger can handle edge cases for all possible player combinations.

StatsLogger():

We will the initialize the StatsLogger multiple times with 6, 9, 12, 15, and 18 robots. We will check the actual output against the expected output, based on our knowledge of the JSON file format. This will ensure our system works for all cases.

updateMatches():

We will call this function on multiple JSON files created by our StatsLogger() tests, confirming that it works in normal game circumstances, as well as exceptional ones. We will check the actual output in the JSON file against expected output to verify correctness. We will also pass in incorrect variables to verify proper error handling.

updateWins():

We will call this function on multiple JSON files created by our StatsLogger() tests for the same reasons as above. We will handle our testing in much the same way as testing for updateMatches().

updateLosses():

We will call this function on multiple JSON files created by our StatsLogger() tests for the same reasons as above. We will handle our testing in much the same way as testing for updateMatches().

updateKilled():

We will call this function on the multiple JSON files created by our StatsLogger() tests for the same reasons as above. We will handle our testing in much the same way as testing for updateMatches().

updateAbsorbed(damage, robotID):

This method will be tested in the same way as updateMatches(), with the exception of passing it damage values that are greater than a robots remaining health to ensure that it properly records actual damage taken and kills the robot. We will also pass in a damage value of 0 to ensure that no damage is taken where applicable.

updateLived(robotID):

Testing is handled the same as in updateMatches().

updateDied():

Testing is handled the same as in updateMatches().

updateMoved():

Testing is handled the same as in updateMatches().

updateExecutions():

Testing is handled the same as in updateMatches().

Gameboard Class

For testing the Gameboard, we will instantiate the GameBoard with a set of gangs. We will run tests to ensure that each robot from each gang is in their correct position at start-up, and then test that the robots are in the correct places after movement (ie. run the updateCoord() function, and then query the coordinate for the robots that are occupying that coordinate. If the robot is not on the coordinate specified, then one of the functions has failed). Because the gangs[] list contains the colour of the robots, it will be trivial as to where each robot starts.

Gamemaster Class

The Gamemaster controls turn order, the calling of the views in the proper order, and when the game ends. The GameMaster class is dependant on its flow of control, and it therefore makes sense to use White Box testing to verify that the correct control paths are taken.

Turns will be tested by generating a list of fake robots and fake gangs and then removing them pseudo-randomly to simulate robot deaths. Testing to verify that gangs with 1, 2, and 3 robots are properly scheduled will occur in this section. We will modify the turn order to print out the next team in the queue, to verify that the scheduling is functional.

The calling of the views in the proper order will be fairly simple, as it is only a matter of placing the code in the correct order. In order to test this sequential functionality before each view is finished, we will create mock views that display sample messages.

StartMenuController Class

This class will be tested during beta testing as it is only used to manipulate the view.

SetupMenuController

To test the SetupMenuController we will provide a sample JSON of all robot's statistics, and test what the output of each public function will be. To test the MakeGangs() function, we will call the function to create a gang, and verify that it is correct by using the GetGangs() function to output what is stored inside the class. All other functions will be tested trivially by comparing their output to that of the information inside of the JSON.

InputController

The InputController will be tested by using the move() and shoot() functions and testing that the changes were applied to the corresponding robot (i.e. its position was updated in the Gameboard, or it was damaged). The concede() function will be tested once the game is complete by having a human player click on it.

EndScreenController

The EndScreenController is responsible for updating the information in the database for each robot at the end of the game. Because of this, the JSON that it is passed will need to be validated (to ensure that it is consistent). All other functions in this class will be tested while using the view.

InterpreterFunctions Class

The functions within the InterpreterFunctions classes fall into three categories. The first group of functions will only alter the internal stack of the InterpreterFunctions class. The second is the group of functions which call corresponding functions from the Robot class as well as checking the internal stack. The third group will alter the forthWords list within the Interpreter class.

Group 1:

These functions will push and pop input from the stack, and testing them will be as simple as checking the pre and post stack state to ensure that it was altered correctly.

String pop():
push(String):
popAndPrint():
drop()
dup()
swap()
rot()
arithmetic()
modulo()
comparison()
and()
or()
invert()
random()

Group 2:

These functions will alter the Robot class. Seeing as the Robot class acts as a storage class, writing a mock class to emulate its functionality would be redundant. This means that Robot should be tested previous to running this classes test code. To test the functions which use the Robot class, values should be popped onto the stack, then the function should be called with a check to ensure that the Robot class was correctly altered. The stack state should also be checked afterwards where applicable. These functions should all run on a test instance of the Robot class with set default values.

health()
moves()
healthLeft()
movesLeft()
attack()
range()
team()
type()
turn()
shoot()
check()
scan ()
identify()
send ()
mesg()
recv()
move(): This function will use the known test robot ID.

Group3:

These three functions alter the List that is held inside of the Interpreter class, and must be called by the Interpreter's play function in the context of a body of Forth code. This testing will be performed inside of the Interpreter class once the rest of the InterpreterFunctions have been tested, as they will be used to show that the logic statements have performed the correct modifications of the List. These functions also require that the play() function from the Interpreter class be tested.

conditionStatement() :

This can be tested by running some simple Forth code through the interpreter which includes a variation of conditional logic statements which alter some value. If the value is correct, then the condition function has succeeded. There should be a body of Forth code which includes multiple if-else branches, as well as nested if statements.

guardedLoop() and countedLoop():

These two functions can be tested by running Forth code which contains multiple forms of each kind of loop. These bits of code should predictably alter the stack so that their success can be easily tracked.

As all three of these functions alter the List in different ways, their interactivity should be tested. To do this, create a piece of Forth code which uses nested loops of each type as well as condition variables. The only place where the individual functions may cause others to fail, is where there is nested

behaviour, or two are run sequentially. As a result, these tests should have a combination of loops and if-statements, both preceding each other and nested inside of each other.

Interpreter Class

The Interpreter class has a high level of coupling with the InterpreterFunctions class. They are in separate classes to allow an interface for the Interpreter's play function, so that it can call a function from the InterpreterFunctions class for each Forth word that it parses. Because these interactions are the main focus of this class, its unit testing will also include testing these interactions.

loadVariables():

Testing this will require a Forth code body which has some variable declarations. The Robot's JSON file can then be manually checked to make sure that the variables were properly declared.

lookUp(String):

This function can be tested by manually adding variables to the Robot JSON file and then calling lookUp and checking to see if it returns the correct value.

play():

This function can be tested by running a piece of Forth code that uses all of the possible Forth commands from the Forth Language document. The block of code should behave predictably so that its output and post stack state can be easily checked. This function should be tested before the conditional and loop functions of the InterpretertFunctions class, but after the rest of its functions.

User Interface Testing

Once the unit and integration testing is complete, we will do an overall system test to ensure the Views are working properly.

StartScreen:

StartScreen():

This will require visual confirmation that all aspects are loaded onto the screen properly. The return values of all built in functions should be tested.

buttonListener():

As there is only one listener for all buttons, each case should be tested.

SetUpMenu:

SetUpMenu():

This testing procedure will be the same as the StartScreen() function

displayRobotSelection():

Tested in the same way as StartScreen()

popupButtonListener():

As there is only one listener for all buttons, each case should be tested.

redrawMap():

This will be tested in the same way as StartScreen(), except that it should be checked for each board size.

GameScreen:

buttonListener():

As there is only one listener for all buttons, each case should be tested.

GameScreen():

Testing will involve making sure that the visible range is properly displayed.

tileListener():

Bound testing should be done with the visible selection range including selecting tiles on the edge and outside of a Robots vision.

redrawSprite():

Tester must ensure that the drawn sprites have proper placement and orientation.

redrawSight():

Ensure that the sight range is properly displayed.

redrawMove():

Ensure that the Robots movement range is correctly displayed.

PassControls:

PassControlsView():

Ensure that the pass controller screen is displayed properly between each player's turn.

EndScreen:

StatsTable():

This function will require an actual comparison of the altered table to make sure that it is correctly altered.

buttonListener():

As there is only one listener for all buttons, each case should be tested.

Integration Testing

To test the integration of all components of the system, a bottom-up approach will be used. Tests will be created for the smallest components first, followed by more tests for each component that uses another. The advantage to this is that errors will be easily traceable throughout the system and that each component may be tested at the time of its creation.

Data Base Testing

The Database functionality will be tested separately from the rest of the system. A mock class will be created which will store a JSON file and simulate the database. All system testing which requires uploading or downloading through the Robot Librarian will use this mock class instead. The database testing will occur in a separate class which will upload and download a document, and check to see that the process worked as expected.

Multiplayer Functionality Testing

The system supports multiplayer functionality, and will therefore require light testing to verify that it functions smoothly for multiple players. Since high data bandwidth during human vs human gameplay is not applicable to our system (due to the “pass the controls” functionality) it will only be necessary to test the AI interactions. Tests will be performed on AI only games, as well as the varying combinations of human(s) vs AI(s). For example, ensuring that the AI does not get stuck in an infinite loop. To test this functionality, group members from our software engineering team will take turns playing the game and verifying all combinations of multiplayer functionality.