Robo Sport Testing Plan

Software Engineering Team: D2

Arianne Butler

Kristof Mercier

Michael Graham

Chris Mykota-Reid

Samuel Horovatin

Sequence:

~~Brief Intro~~

~~Objectives / tasks~~

~~Scope~~

~~Test Targets~~

* + ~~Identify and list any use-cases, objects, functionality, etc. that have been identified as test targets~~
  + ~~Examples of target areas:~~
    - ~~Data and Database Integrity Testing~~
    - ~~Unit Testing~~
    - ~~GUI Testing~~
* Test Strategy
* Resources used for testing
* References (if applicable)

# 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.

# 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 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 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, it 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 we know the structure of the JSON, we can then use that to determine what the output of all of the public functions of the robot should be. For example: getTeam() should be the software team’s ID that is stored in the JSON, 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 can just get a robot to send multiple messages to itself, and then get it to check its own inbox. If the messages are returned in the correct order (ie. the first message sent is the first message received) then you know that 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 we can query the system to check if the robot’s health has been decreased by the correct amount, and we can query if the robot can shoot again (which it shouldn’t be able to).

### Gang Class

Testing the Gang class will be fairly straight forward. Similarly to the robot class, it will just be checking to ensure that the input into the class is the same as the output. As the information is never changed once a Gang is created this means that it should always be the same.

### StatsLogger Class

We have to make sure the Stats Logger properly stores and then sends the data it reads from its function calls since stats logging is a key part of our system requirements.  Making sure it works is therefore an important thing to do.  The testing method will be black box testing since most of the functions are very simple in their implementation and don’t have tricky control logic that you’d normally want to white box test for.  We will give it a bunch of phony inputs and then check to make sure that Stats Logger properly returns a JSON file and that the values are correct (based on the phony inputs and our knowledge of how JSONs are supposed to look).  Each function will have to be unit tested and the overall class will have test cases for 0 robots, 1 robot, 2 robots and 9, 12, 15 and 18 robots.  This will be to ensure that the Stats Logger has the edge cases and all possible number of robots in a run of a game tested.

StatsLogger():

we will the initializer it multiple times with 0, 1 ,2 , 6, 9, 18 and some arbitrarily large number of robots.   We will check the output against what we expect the output to be knowing the format of JSON files.  This will ensure our system works in the normal use cases as well as giving us more assurance that there are no bugs at all.

updateMatches():

We will call this function on the multiple JSON files created by our StatsLogger() tests once again making sure it works in normal game circumstances and exceptional ones for assurance of stability.  We will check the JSON file against expected output to verify this test. We will also pass it incorrect variables to make sure it handles them correctly throwing the proper exceptions.

updateWins():

We will call this function on the multiple JSON files created by our StatsLogger() tests for the same reasons as listed above. We will check it in the way same too.

updateLosses():

We will call this function on the multiple JSON files created by our StatsLogger() tests for the same reasons as listed above. We will check it in the way same too.

updateKilled():

We will call this function on the multiple JSON files created by our StatsLogger() tests for the same reasons as listed above. We will check it in the way same too.

updateAbsorbed(damage, robotID):

tested in the same way as updateMatches() with the exception of passing it damage values greater than a robots remaining health to ensure it properly records actual damage taken and not over kill as well as no damage to make sure it properly records no damage being taken.

updateLived(robotID):

same testing method as updateMatches().

updateDied():

same testing method as updateMatches().

updateMoved():

same testing method as updateMatches()

updateExecutions():

same testing method as updateMatches().

### Gameboard Class

For testing the Gameboard we will instantiate the gameboard with a set of teams. Tests will be run to ensure that each robot from each gang is in their correct position at start-up, and then tests will be run to move the robots (ie. run the updateCoord function, and then query the coordinate for the robots that are on it. If the robot is not on the coordinate specified, then one of those functions have 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 is in control of the turn order, calling of views in the proper order and when the game ends and so has to be able to do all of these things properly.  For the Gamemaster it makes more sense to use white box testing as the process of the Gamemaster is very dependant on the control paths it takes and we would be testing to make sure it goes down the right ones.

The turns will be tested by generating a list of fake robots and fake gangs and then removing them pseudo-randomly to simulate deaths of robots.  Testing to make sure gangs with one robot, two robots and all 3 robots get scheduled properly will be the focus here.  We might modify turn order, at first, to print out the next team in the order to verify that it works.

The calling of the views in proper order will be a fairly simple test since it is a matter of just placing the code in the right order.  Without having to get every view coded and working before testing the Gamemaster we can create mock views that display simple messages in place of the full view.

### StartMenuController Class

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

### SetupMenuController

To test the SetupMenuController we will be providing a sample JSON of all robot’s statistics, and from there we will test what the output of each public function will be. To test the MakeGangs() function we will use it to create a gang, and verify that it is correct by using the GetGangs() function to output what was stored inside of 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 seeing if the changes were applied to the corresponding robot (ie. its position was updated in the Gameboard or it was damage). The concede() function will be tested once the game is complete by getting a player to 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 if 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 functions 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 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 are when there is nested behaviour, or two are run sequentially. Because of this these tests should have a combination of loops and if statements preceeding 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 just 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 looked up 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

## Integration Testing

To test the integration of all components of the system a bottom-up approach will be used. That is, tests will be created for the smallest components and following that more tests will be run for each components that uses one 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 their creation.

## Data Base Testing

The Database functionality will be tested separately from the rest of the system testing. 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. 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.