TEST PLAN

Project Name: RobotGo

Team: CMPT370-B3

Team Member: Cheng, Gong
Hounjet, Carlin
Lan, Shaoxiong
Xie, Joey
Yue, YiRui

Date: Nov 06, 2016

Table of Contents

Part 1: Introduction

Welcome to our document detailing the testing plan for group B3,aka group ROBOT GO! Here you will find an outline of how our test driven design plan will unfold during development, followed by a plethora of testing suites for all of our major classes and their interactions. What isn't included is tests for checking setters and getters and other minor functions and classes that are self explanatory in their behaviour. What we did find it necessary to include are test suites for our controller classes (initialization and the main game controller) as well as a detailed test plan for the robot interpreter. We kept the plans separate and distinct because these classes can be developed independently through unit testing and functional testing up until integration becomes possible due to our architecture detailed in our design document.

This document begins with the unit tests for all classes, followed by functional testing, integration testing and ending with a plan for a full system's test. Each of these steps are in order of Main game controller, initialization controller and ending with tests for the robot interpreter. We begin with the main game controller because it is the most important to tie the system together. Without it being properly tested and implemented, the initialization controller and robot interpreter will never be able to be fully tested and implemented, therefor it is best to get an idea of how it works before moving on to test the other classes.

Despite the initialization controller's behaviour coming first chronologically when the system is run, it's tests are dependent on being able to set things up in the main game controller.

The robot interpreter is the final thing we test because in our architecture, it is basically a self contained system and should be thought of as a separate develop- ment plan until it reaches the point where it is ready to communicate with the main game controller. The other two can function without it and would even be able to run Player vs Player games without the robot interpreter integrated into the final system.

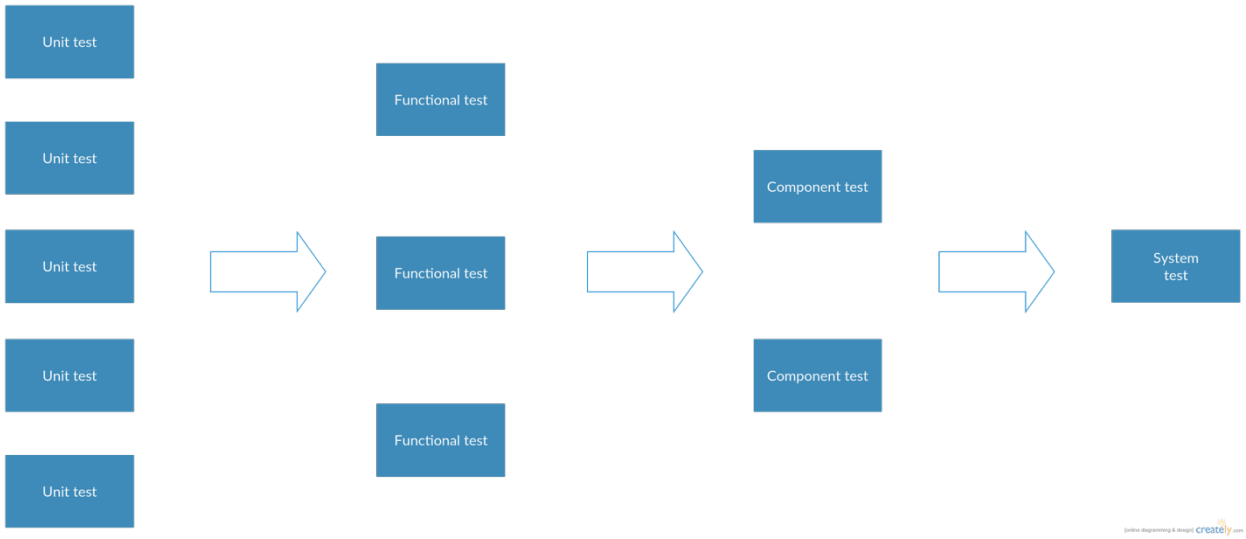
Our least robust level of testing is our system's test, as it follows the basic lo- gic of game testing and there isn't a whole lot to say about it beyond detailing the different modes that we plan to playtest. This may involve bringing in new users to try and break our system, or we may just test

it ourselves.

On to the outline!

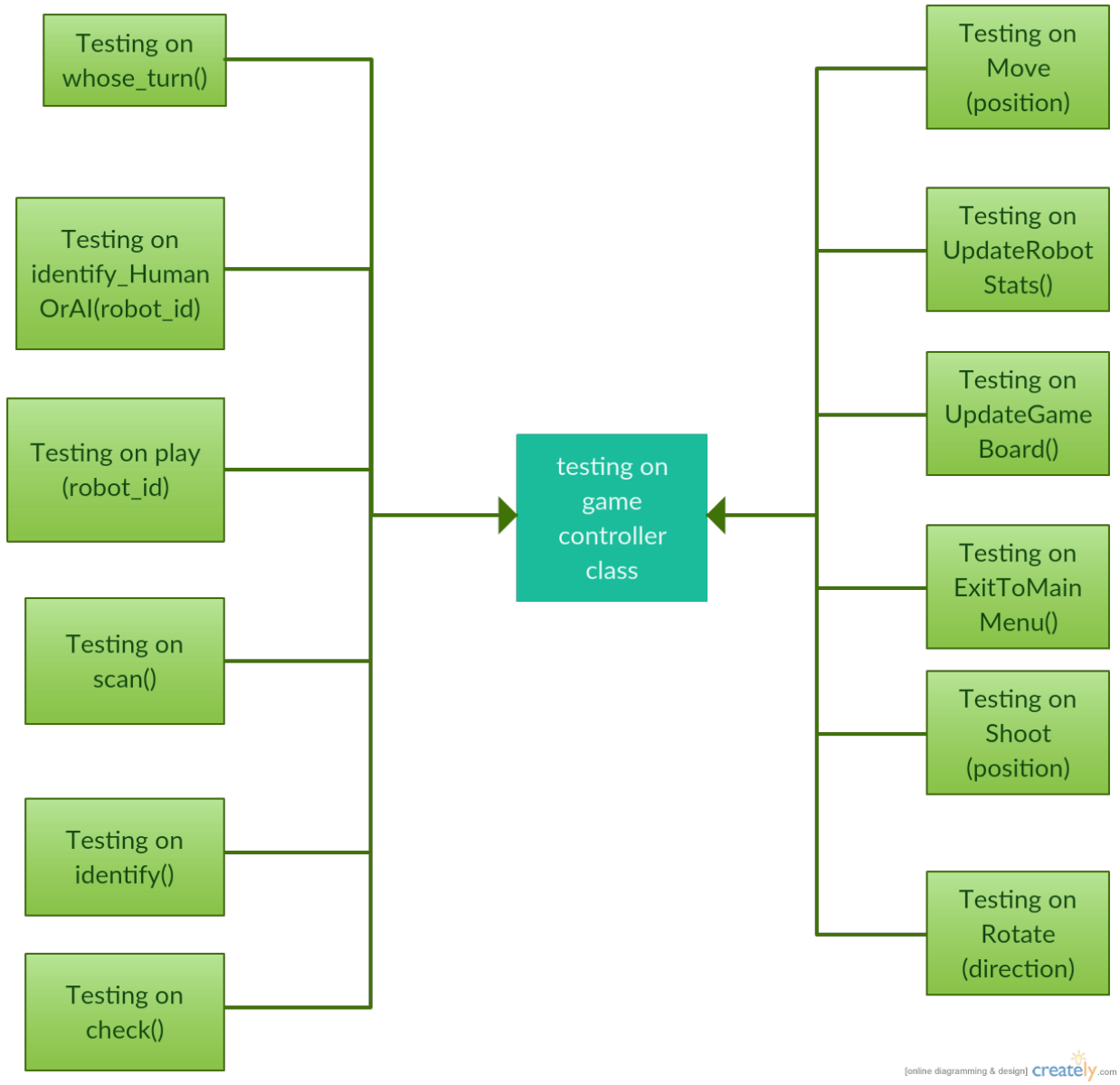
Part 2: Test Diagram

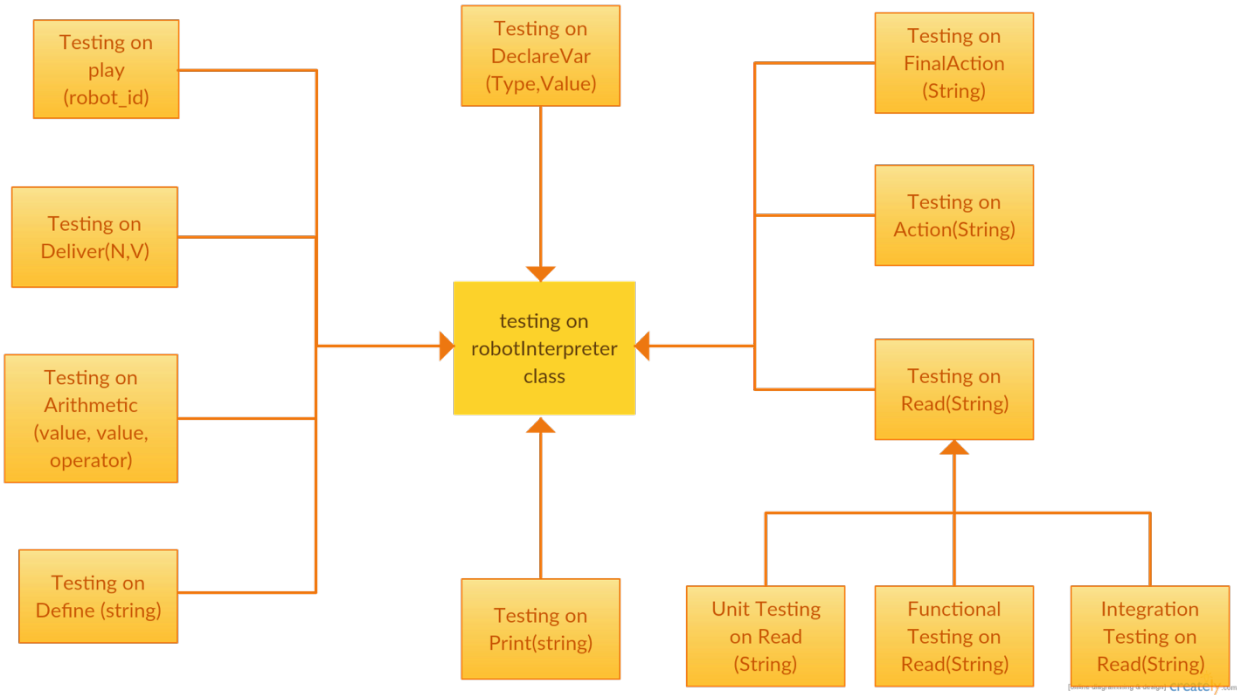
2.1 The structure used for our progression of testing

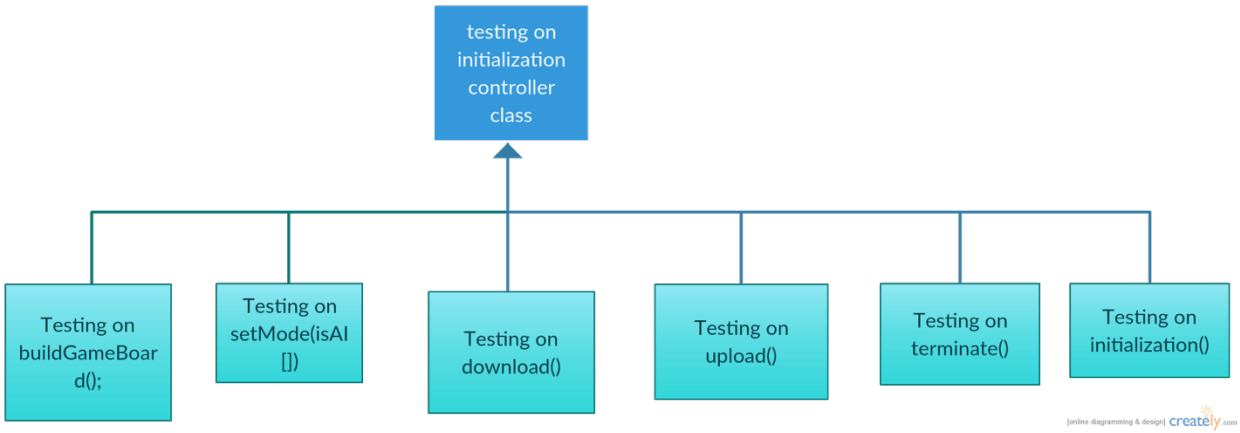


2.2 An example of how our testing evolves

2.3 All the function tests we have, classified by the different classes the function belongs to.




Testing on play (robot_id)
Testing on Deliver(N,V)
Testing on Arithmetic (value, value, operator)
Testing on Define (string)
Testing on DeclareVar (Type,Value)
testing on robotInterpreter class
Testing on Print(string)
Testing on FinalAction (String)
Testing on Action(String)
Testing on Read(String)
Unit Testing on Read (String)
Functional Testing on Read(String)
Integration Testing on Read(String)


testing on initialization controller class
Testing on buildGameBoard();
Testing on setMode(isAI[])
Testing on download()
Testing on upload()
Testing on terminate()
Testing on initialization()

Part 3: Unit Tests of important functions

3.1 Unit Tests for functions inside gameController class

3.1.1 test_whose_turn()

To test the whose_turn inside the game controller class.
Get information stored in robot class. Based on the the speed of each robot, identify the turn of the robot in different teams.

- **Case 1:** The game is not initialized, which means no teams and no robots created.
  getMaxSpeed();
  throw exception "getMaxSpeed need team name as input";

- **Case 2**: All robots are still alive. Suppose trying to find the robot trun in Team A. Suppose the robot_id of the maxmium speed in TeamA is 1001.
  int result = maxgetMaxSpeed(TeamA[]);
  int expectResult = 1001;
  AssertEquals(result, expectedResult);

- **Case 3:** Some robots are shot dead in teamA[].
  TeamA[i] = maxgetMaxSpeed(TeamA[]);
  if isAlive (TeamA[i]) is ture
  return robot_id of TeamA[i]
  else
  getMaxSpeed(TeamB[]) -- skip to next team, suppose the robot_id of the maxmium speed in TeamA is 1004.
  getMaxSpeed(TeamA[])
  int result = maxgetMaxSpeed(TeamB[]);
  int expectResult = 1004;
  AssertEquals(result, expectedResult);

3.1.2 test test_identify_HumanOrAI(robot_id)

To test the identify_HumanOrAI(robot_id) inside the game controller class.

- **Case1:** The game is not initialized, which means no teams and no robots created.
  identify_HumanOrAI(robot_id)

throw exception “no robot_id is provided”

- **Case 2:** It’s the human players’ turn
  If the robot belongs to the human team, pass the robot_id as the input to the play(robot_id) function inside the game controller.

- **Case 3:** It's the AI's turn
  If the robot belongs to the AI team, pass the robot_id as the input to the play(robot) function inside the interpreter.

3.1.3 test_play(robot_id)

To test the play(robot_id) inside the game controller class.
Note: this function will call other functions within the same class, and the function from another class.

- **Case 1:** The game is not initialized, which means no teams and no robots created.
  play();
  throw exception “no robot_id is provided”

- **Case 2:** The given robot_id is not valiad
  play(-123);
  throw exception “the given robot_id doesn't exists”

- **Case 3:** Given a valid robot_id which belongs to the human team, and the human team chooses to move 1 step from positionindex 1 to postionindex 2.
  play(1001); // this will automatically call rotate() and move() in side the same calss
  int newpostion = getPosition(1001);
  int expectedResult = 2;
  AssertEquals(newpostion, expectedResult);

- **Case 4:** Given a valid robot_id which belongs to the AI team, and the AI team chooses to shot the robot dead.
  play(1004); // call play() in interpreter
  bool lifeStatus = isAlive(1004);
  bool expectedResult = false;
  AssertEquals(lifeStatus, expectedResult);

3.1.4 test_scan()

To test the scan() inside the game controller class.

- **Case 1:** The game is not initialized, which means no teams and no robots created.
  scan();
  throw exception “the game is not initialized”

- **Case 2:** Return the right numbe of enemies on game board, if the number of enemies is more than 0.
  scan();
  int result = scan();
  int expectedResult = 2;
  AssertEquals(result, expectedResult);

- **Case3:** the number of enemies is 0.
  int result = scan();
  int expectedResult = 0;
  AssertEquals(result, expectedResult);

3.1.5 test_identify()

To test the identify() inside the game controller class.

- **Case 1:** The game is not initialized, which means no teams and no robots created.
  identify();
  throw exception “the game is not initialized”

- **Case 2:** Return the right numbe of enemies on game board, if the number of enemies is more than 0.
  identify();
  int result[] = identify();
  int expectedResult[] = [red, 2, 1, 2];
  AssertEquals(result, expectedResult);

- **Case3:** the number of enemies is 0.
  int result[] = identify();
  int expectedResult[] = [red,0,0,0];
  AssertEquals(result[], expectedResult[]);

3.1.6 test_check()

To test the identify() inside the game controller class.

- **Case 1**: The game is not initialized, which means no teams and no robots created.
  check();
  throw exception "the game is not initialized";

- **Case 2**: The adjacent of a given direction is occupied.
  check();
  bool result = isOccupied;
  bool expectedResult = true;
  AssertEquals(result, expectedResult);

- **Case 3:** The adjacent of a given direction is NOT occupied.
  check();
  bool result = isOccupied;
  bool expectedResult = fasle;
  AssertEquals(result, expectedResult);

3.2 Unit Tests for functions inside robotTranslator class (Interpreter)

3.2.1 test_Read()

We will start simple and test Read()'s ability to take in a string and break it down based on Forth Syntax while ignoring comments. This will be done as a black box test by writing a string with a comment and words delimited by spaces then popping them from the stack to be printed onto the console (via the word ".") for inspection. If a word is read that isn't in the dictionary, an error message is displayed on the console and read() fails.

- **Case 1**

Preconditions: The function for the word "." is defined and works as expected, printing whatever pops off the data stack onto the console.
Input: "(this is a test) ."This" . ."is" . ."a test" . 5 ."
Output to command line: This Is a test
5

- Case 2
  Input " word "
  Output: Fatal failure, word "word" is not defined, ending read()

3.2.2 test_Deliver (N,V)

To test the Deliver(N,V) inside the robotTranslator class.

- **Case 1:** the message sent to teammate N is null
  Deliver(N,V);
  String result = getMessage(N);
  String expctedResult = null;
  AssertEquals(result, expectedResult);

- **Case 2:** the message sent to teammate N is "shot!"
  Deliver(N,V);
  String result = getMessage(N);
  String expctedResult = "shot!";
  AssertEquals(result, expectedResult);

3.2.3 test_play(robot_id)

To test the play(robot_id) inside the RobotTranslator class.

- **Case 1:** The game is not initialized, which means no teams and no robots created.
  play();
  throw exception "The game is not initialized"

- **Case 2:** Given a valid robot_id which belongs to the AI team, and the AI team chooses to shot the robot dead.
  play(1004); // call play() in interpreter
  bool lifeStatus = isAlive(1004);

bool expectedResult = false;
AssertEquals(lifeStatus, expectedResult);

- **Case 3:** Given a valid robot_id which belongs to the AI team, and the AI team chooses to move 1 step from positionindex 1 to postionindex 2.
  play(1001); // this will automatically call rotate() and move() in side the same calss
  int newpostion = getPosition(1001);
  int expectedResult = 2;
  AssertEquals(newpostion, expectedResult);

3.2.4 test_Arithmetic (value, value, operator)

- **Case 1**: the inpit is not valid.
  Arithmetic (5, 0, /);
  throw exception “the dividend can not be 0”

- **Case 2:** the input is valid
  int result = Arithmetic (5, 2, +);
  inr expctedResult = 7;
  AssertEquals(result, expectedResult);

3.2.5 test_Define (string)

- **Case 1:** the input string "shot" is not defined
  Define(shot);
  bool result = isDefined(shot);
  bool expectedResult = false;
  AssertEquals(reslut, expectedResult);

- **Case 2:** the input string "move" is already defined
  Define(move);
  bool result = isDefined(move);
  bool expectedResult = true;
  AssertEquals(reslut, expectedResult);

3.2.6 test_DeclareVar (Type,Value)

- **Case 1:** the varaible lifePoints is not defined
  DeclareVar (lifePoints, 2);
  int result= getlife();
  int expectedResult = 2;
  AssertEquals(reslut, expectedResult);

- **Case 2:** the varaible lifePoints is already defined
  DeclareVar (lifePoints, 3);
  int result= getLife ();
  int expectedResult = 3;
  AssertEquals(reslut, expectedResult);

3.2.7 test_Print(String)

Takes in string off the Data stack after "." word is called

- Case 1: to print a valid string
  Print("."This.."is".."a test". 5.")
  Expected output looks like:
  This
  Is
  a test
  5
- Case 2: to print a string without "."
  Print("This is a test")
  throw exception "invalid input"

Part 4: Functional Tests

In this part we test that, when putting individual unites together, if they will work as a functional unit. Since the number of testing involved in this part will be massive, we choose the most important unit **read(), and all other units related to read()** to complete the functional test.

We can then expand upon this base by using our test function to input Forth code strings that test each of our Forth word's functions (Stack words, Arithmetic words, Comparison words, Logic and Control words, Loop words, Status Words) and read()'s ability to recognize them.

- **Case 1** Testing stack control words by pushing values onto the stack then modifying and printing them to verify changes
  Input: “1 drop 2 dup . 3 swap . . ”
  Output:
  2
  2
  3
- **Case 2** Testing all arithmetic words by calling them then printing the value to console.
  Input: “ 1 1 + . 4 3 - . 2 5 * . 47 5 /mod . . ”
  Output:
  2
  1
  10
  9
  2
- **Case 3** Test read()'s ability to declare, store, and retrieve variable words.
  Input: “ variable test1 ;
  variable test2 ;
  variable test3 ;
  variable test4 ;
  : test1 1 ;
  : test2 .“test 2” ;
  : test3 true ;
  : test4 if test3 test1 . test2 . else .“comparison failed” then . ;
  test4

“

Output:
1
test 2

- **Case 4** Testing Comparison words

“

```
1 1 = if ."equal" else ."unequal" then .
2 1 = if ."equal" else ."unequal" then .
1 1 <> if ."the same" else ."different" then .
2 1 <> if ."the same" else ."different" then .
.”same” .”same” <> if ."the same" else ."different" then .
.”same” .”different” <> if ."the same" else ."different" then .
1 1 <= if ."less than or equal" else ."greater than" then .
2 1 <= if ."less than or equal" else ."greater than" then .
1 2 <= if ."less than or equal" else ."greater than" then .
1 1 => if ."Greater than or equal" else ."less than" then .
2 1 => if ."Greater than or equal" else ."less than" then .
1 2 => if ."Greater than or equal" else ."less than" then .
1 2 < if ."less than" else ."not less than" then .
2 1 < if ."less than" else ."not less than" then .
1 1 < if ."less than" else ."not less than" then .
2 1 > if ."greater than" else ."not greater than" then .
1 1 > if ."greater than" else ."not greater than" then .
1 2 > if ."greater than" else ."not greater than" then .
```

”

Expected Output:
Equal
Unequal
the same
different
the same
different
less than or equal
greater than
less than or equal
Greater than or equal

Greater than or equal
less than
less than
not less than
not less than
greater than
not greater than
not greater than

- **Case 5**
  Testing nested loops. "I" should push the index value for each different loop. Invalid parameters on loop will result in a fatal error that will end read()

  Input: "0 1 do I . 1 5 do I . loop loop "
  OutPut:
  0
  1
  2
  3
  4
  5
  1
  1
  2
  3
  4
  5

- **Case 6** testing leave word within loops
  Input: "0 5 loop dup I . 3 = if leave else loop then"
  Output:
  0
  1
  2
  3

- **Case 7** Testing guarded loops

  Input:

  “

  : variable Count ;

  begin

  Count 1 + count !

  Count .

  5 Count = until ”

  Output

  1

  2

  3

  4

  5

Part 5: Integration & Component Test for important functions

In this part, we test the functions in one class that will work with other classes.

5.1 Tests related to robotTranslator class

Before read() and it's class RobotInterpretor can be integrated into the system, we must ensure that it can send and receive function calls using action words (check!, scan!, identify!, send!, mesg!, reciv!) and final action words (move! and shoot!). We will do this in our test function by populating the robot's variables and inputting strings calling for these actions while checking to see if the function calls are returned. We will have to have the words call fake functions only for testing, which will be changed to calls and returns from Game controller as the rest of the system is completed and integrated into the code.

5.1.1 To test check!

We need to see if read() recognizes it, runs the function, then pushes the expected return value onto the stack.

Preconditions: Either there is a test function that has built a mock tile to check where direction 0 is empty, direction 1 is occupied, and direction 3 is out of bounds or gamecontroller has set up that situation.

- **Case 1**
  Input
  0 check! .
  1 check! .
  2 check! .
  Output:
  EMPTY
  OCCUPIED
  OUT OF BOUNDS

- **Case 2** Somehow an invalid integer or invalid token is popped into the Check parameter
  Input: 7 check!
  Output:

ERROR integer out of range for check!

- **Case 3**
  Input:
  "."hi" check!"

  Output:
  ERROR invalid input for check!

5.1.2 Testing scan!

With a test function that returns the value "3" when scan is called, simulating the return value for scan! If there were 3 robots on the board. Once the rest of the system is complete, we can populate a game board with three robots in various spots to test this function further

Input: "scan!"
Output: 3

5.1.3 Testing identify!

- **Case 1**
  With a test function that returns the expected values of a robot on the blue team at range 2 in direction 1 with remaining health of 2. Or a gameboard with that scenario set up
  Input:
  "1 identify! . . . . "
  Output:
  BLUE
  2
  1
  2

- **Case 2**
  Identify is called but there is no robots to identify
  Input:
  "1 identify! . . . . "
  Output:
  ERROR there were no robots to identify

5.1.4 Testing send!, mesg!, reciv!

With either a test function that saves messages in the mailboxes, or when the game controller is complete and can do this function. If any of these commands recieves invalid parameters (anything that isn't a SNIPER, TANK, or SCOUT) then it's a fatal error and read() stops.

- **Case 1** Send a mail message to it's self (assume robot is SNIPER) then check for it and receive it and print it onto the console
  Input: " ."SNIPER" ."Hello me!" send! ."SNIPER" mesg! If ."SNIPER" reciv! Else ."No mail" . Then ."
  Output: Hello me!
- **Case 2** Don't send a message to self then check for mail
  Input: "."SNIPER" mesg! If ."SNIPER" reciv! Else ."No mail" . Then ."
  Output: No mail
- **Case 3** reciv! Is called when there is no mail. This should return nothing
  Input:
  " ."SNIPER" reciv! "
  Output: nothing

5.1.5 Testing Move! Rot! And Shoot!

This would simply consist of test functions that take in values to check for validity of parameters until the Game Controller is complete and the actions are verified there since there are no return values.

5.2 Tests related to the GameController class

5.2.1 Move(position)

Move should not do anything if there is not enough movement points
It should not do anything if an invalid position is used. The valid values of position are 0-60 if it is a board of size 5 or 0-126 if it is a board of size 7. After the move the robot should be in the position that was given.

An instance of the robot class and the gameboard class is required for these tests. The rotate method should also be implemented before this method.

- Case1: method should throw an exception if the position is invalid.

- Case 2: method should create a messagebox "not enough movement points" if the distance of the target tile is greater than the moves left of the robot. The robot should not move after this.

- Case 3: the robot should appear in the specified position after move is executed if there is no error.

5.2.2 Shoot(position)

Robot should not be able to shoot if it has already shot this turn. It should not be able to shoot if the target is not in range. The valid range of targets is 0-60 for a board of size 5 or 0-126 for a board of size 7. Robot must be an instance of the robot class.
An instance of the robot class is required for these tests.

- Case 1: method should throw an exception if the position is invalid.

- Case 2: method should create a messagebox "already shot" if the robot has already shot this turn.

- Case 3: method should create a messagebox "out of range" if the target tile is not in range.

Multiple instances of the robot class are required for the tests below.

- Case 4: check that the shotThisTurn variable in robot is changed to true if the robot shoots.

- Case 5: if there is no robot on the targeted tile then a message box will be created "no one there"

- Case 6: check that the shot robot is damaged by checking the hp of the affected robot.

- Case 7: check that the method works for shooting multiple robots on the same tile including the case where the shooter is on the targeted tile.

5.2.3 Rotate(direction)

The robot should be in the right direction after turning.
The valid values of direction are 0-5. Robot must be an instance of the robot class.
These tests require an instance of the robot class

- Case 1: The method must throw an exception if the direction is invalid.

- Case 2: Execute the method rotate on a robot to change its direction to an arbitrary value and check if the robot had its direction changed.

5.2.4 UpdateRobotStats

When a robot is damaged damageTaken should increase by the amount of damage. When it dies the amount of deaths increases by one. When it kills the number of kills increases. When it moves tiles moved should increase by the number of tiles moved this turn. When it inflicts damage it should the damagedInflicted should increase appropriately.

These tests require the gameboard view, gameboard model, and instances of the robot class. Shoot,move and rotate should be implemented before this method is tested.

- Case 1 : when a robot is damaged check the damage taken.
- Case 2 : check that the robot deaths increased after death.
- Case 3 : check that the kills increased after killing.
- Case 4: check that the tilesMoved increases after moving.
- Case 5: check that damagedInflicted increases after inflicting damage
- Case 6: repeat test 1,2,3, and 5 with multiple robots being killed or damaged instead of just one.

5.2.5 UpdateGameBoard

The gameboard must show: dead robots removed, changed ui, moved robots at their destinations, shot robots damaged and the current turn round and robot.

These tests require the gameboard view, gameboard model, and instances of the robot class. Shoot,move and rotate should be implemented before this method is tested.

- Case 1: gameboard must be displayed in the gameboard view
- Case 2: check that robots are removed from view after being killed.
- Case 3: check that the ui showing the robot hps are being updated when the robots are damaged
- Case 4: check that the current turn and round are updating,
- Case 5: check that the current robot is indicated.

5.2.6 ExitToMainMenu

Main menu view should be displayed and the gameboard view should not. Check that the correct view is shown. This test requires both views.

Part 6 System Test

Tests involves the initialization controller should be deemed as system tests.

6.1 Test for buildGameBoard();

Building the game board involve multiple classes so this is a system testing.

- **Case 1**
  Input: the size 5 along with the other values are correspondingly inputted and the game successfully starts.

  Output: there is a hexagon board of each side with 5 tiles is shown in the main game interface.

  This test can only be checked by viewing.

- **Case 2**
  Input: the size 7 along with the other values are correspondingly inputted and the game successfully starts.

  Output: there is a hexagon board of each side with 7 tiles is shown in the main game interface. The test can only be checked by viewing.

- **Case 3**

  Input: the input value is missing.
  Output: the game should throw an exception

6.2 Test for setMode(isAI[])

- **Case 1**
  Input: when all the robot was chosen to be AI
  Output: The game should be able to go on without any involvement of human. This test can only be done by actually playing, so this test is also part of system testing.

- **Case 2**
  Input: when some are robots and some are AI

Output: when it is human's turn to play, the play function should be enabled and when it is robot's turn to play, the robot should be able to respond correspondingly and quickly.

- **Case 3**
  Input: when all human is chosen.
  Output: the play function should be always enabled for the current player.

6.3 Test for download()

This test involves classes robot and download so it is a system test.

- **Case 1**
  Input: there is no robot script.
  Output: throw an exception.

- **Case 2**
  Input: there are more robot script than needed.
  Output: throw an exception

6.4 Test for upload()

This test involves classes robot and upload so it is system test.

- **Case 1**
  Input: the color and number of robot team is chosen
  Output: the script is successfully uploaded and the data exactly matches what are chosen.

- **Case 2**
  Input: one of the color and number of robot team is missing
  Output: throw an exception.

6.5 Test for terminate()

This is a system test and it can only be tested after the game can run.

Input: the terminate button is clicked.
Output: the game ends successfully.

6.6 Test for initialization()

Input: all the data needed are inputted appropriately and correctly.
Output: the game board is appropriately built and robots are appropriately loaded.

Part 7 Summary

Since we chose the model view controller architecture as the main structure for our project, all of the intelligence of the code is concentrated in the classes of our controllers. The controller also has interactions with the model and the view. In our testing plan we have included unit and functional tests of the controller methods. We also have written integration tests for the interfaces between the controller, the model and the view.

Another area of testing in our project is the RobotInterpreter. Because of the complexity of the responsibilities the interpreter has, we have decided that it was necessary to write a suite of tests for this component as well. We have made unit and function tests for the interpreter as well as integration tests for the places the interpreter interfaces with the rest of the system. We have made some changes our design and these changes are enumerated at the back of our document.

Changes:

1. We discussed separating the read() function from the robot interpreter since it could easily be brought out and given it's own methods, simply passing function calls to the robot interpreter and back again. This didn't effect the tests we wrote so this is a non final change.

2. We also discussed how building robots for AI may have to happen in robot interpreter, but this also didn't effect our testing so we will make these changes if necessary during implementation