CMPT 370 D1

TEST PLAN

Date: Nov 5, 2016

Table of Contents

1 Introduction

This Test Plan has been created to communicate the test approach for our system to team members. It includes the objectives, scope, schedule, risks and approach. This document will clearly identify what the test deliverables will be and what is deemed in and out of scope.

1.1 Objectives

The group will be using JUnit testing to test the functionalities of the Robot Game. JUnit is a unit testing framework for the java programming language and it works well to implement the Test-Driven Development process the group will use the test their code. The testing team in the group will be all the included group members.

1.2 Team Members

Resource Name	**Role**
Dimaano, Nico	Developer / Tester
Neijmeijer, Niklaas	Developer/ Tester
Seidenthal, Kyle	Developer / Tester
Sterma, Brendon	Developer / Tester
Zang, Jiawei	Developer / Tester

2 Risks

2.1 Risks

The following risks have been identified and the appropriate action identified to mitigate their impact on the project. The impact (or severity) of the risk is based on how the project would be affected if the risk was triggered.

#	**Risk**	**Impact**	**Mitigation Plan**
1	Team members lack the required skills for JUnit testing	High – to schedule and quality	Plan training course to skill up.
2	The project schedule is too tight, and there is not enough time to test the project	High	Set test plan priority, or test core functionality rather than test all.
3	Hardware Failure	Medium	Use version control on a remote server

3 Changes

3.1 Changes in Model

The major changes to the model section was shuffling some methods and attributes between the User, Player and Robot classes, as well as the introduction of the Board class, which is used to keep track of what Tile is where.

3.2 Changes in View

All the changes to the view classes were syntax corrections.

3.3 Changes in Control

The Game class has mostly been reworked, it now has methods to handle all the inputs from the view, and manage the turn to turn running of the game itself.

4 Test Approach

Our team is using a waterfall approach to develop the Robot game. The Requirements and Design/Architecture step has been finished and the test team may start creating tests. Exploratory testing will play a large part of the testing as the group has never used this type of tool and will be learning as they go. Tests for planned functionality will be created and added to the JUnit testing code collection as we get iterations of the product.

4.1 Test Details

4.2.1 Model

The model contains the game logic and entities for the Robot Game. This section will be tested by automation using J-Unit testing. Below are the details on the classes in the model and how we plan to test them.

Board: The Board class will represent the game board in the Robot Game. It contains a 2D Array of tiles and a list of players on the board. It also has methods movePossible and attackPossible which determines if the action selected by the player can be done.

movePossible: The move possible function determines if it is possible for a specific robot to move to a specific tile by checking that the robot's remaining movement score is equal to or greater than the distance from the current tile to the given tile. This function will be tested by checking if there is a possible move between the first robot and the second tile, which should

return true. We will also check that moving the second robot to the second tile returns false, as it is not in range.

Pseudocode

Create a robot
Place it on a tile
Create a tile in the range of the robot
Assert(movePossible(robot, destination))
Create a tile out of the range of the robot
Assert(!movePossible(robot, destination))

attackPossible: The attack possible function determines if it is possible for a given robot to attack a given tile by checking if the given robot's range value is equal to or greater than the distance from the current tile to the given tile, and the given tile contains a robot. It returns a Boolean value. We will test this function by checking if the first robot can attack tile 1, which it should be able to. We will then check that the first robot cannot attack tile 2.

Pseudocode

Create a robot on a tile in range of the attacking robot
    Assert(attackPossible(attacker, enemyTile))
Create a robot on a tile out of range of the attacking robot
    Assert(!attackPossible(attacker, enemyTile))

User: The User class represents a user and their attributes in the game. Most of the functions in this class are getters and setters, and are too trivial to spend time testing. One function, inspectTile, will be tested as below.

inspectTile: The inspect tile function allows the user to obtain information about the selected tile. We will test this by creating a tile and robot. We will put the robot on the tile, and call the inspectTile method on that tile. The results will be compared to the expected output, which is the robot on the tile.

Pseudocode

Create robot
Create tile
Put robot on tile
inspectTile(tile)
assert(robot_is_returned)

Observer: The Observer class is a subclass of the User class. It can watch a game in progress, and leave the game. Because this class inherits methods from the User class, we need only test one method as follows.

leaveGame: The leave game function will return the observer to the main menu of the game and disconnect them from the match. We will test this function by connecting to a fake match, calling leaveGame, and checking the server status.

Pseudocode

```
Observer.leaveGame();
Assert(Game.checkServerStatus() = 0)
```

Player: The Player class is another subclass of the User. The Player can move robots, attack tiles, end their turn, and forfeit the game. Each of these actions has a function related to it, and each will be tested as follows.

moveRobot: The move robot function allows the player to move a given robot to a given tile. It will be tested by creating a robot and placing it on a tile. We will then create a destination tile and check that it is possible to move to that tile. Then the move robot function will be called and it will be verified that the robot is now on the destination tile. We will also check that the robot's remaining moves counter has been properly updated.

Pseudocode

```
Store how many moves the robot has left
        Check that it has more than 0 moves left
        Store which tile the robot starts on
                Confirm that it is possible to move between the
            starting and ending tile
            If both are true
            Move the robot
                Confirm that the robot is now on the ending tile
                Store the x and y values of the new tile

                Confirm that the tile in the board thinks that the
                robot is where it should be

                Confirm that the robot has one less move left
```

attack: The attack function allows the player to attack a tile with one of their robots. This will be tested by first checking that the player has not attacked yet this round, which they have not. We will then confirm that an attack from

the robot's tile to the destination is possible. After this, we will attack another tile with a robot on it, and compare the attacked robot's health to what is expected. It is expected that the attacked robot's health will be its original health minus the damage of the attacking robot.

Pseudocode

Confirm that the player has not made an attack this turn.
Confirm that an attack from the robots space to the destination is possible.
If both are true
    Store the list of robots on the target tile.
    Execute the attack.
    Compare the stored Robots health with the Robots left on the tile to confirm that the correct damage has been done.
    Confirm that the player has made an attack this turn.

forfeit: The forfeit function allows a player to quit the game during an active match. The player will then become an observer. This will be tested by ensuring that the player's robots are removed from the board and creates an observer with the same name as the player.

Pseudocode

Store the player's list of robots
Have the player forfeit
Confirm that the player's robot has been removed from the tiles they occupied.
Confirm that the player is now an observer

Server: The Server class is a class that will handle all networking between players. It is responsible for making sure all players have the most recent game state.

updateGameState: The update game state function sends the current game state from one player's machine to all other player's machines. This will be tested by creating a connection and sending the current game state across it. The received version of the state will be compared to the sent version to ensure they are the same.

Pseudocode

Create Connection
Create game state
Send game state
Assert(received_state = = sent_state)

connectToHost: The connect to host function will connect a machine to the specified host. This will be tested by creating a fake host and attempting to connect to it. We will then check the connection by sending a small test message.

Pseudocode

Create host
connectToHost(host)
assert(connected)
send test message
assert(message_recieved)

disconnectFromHost: The disconnect from host function will disconnect the player from the host they are currently connected to. We will test this by creating and connecting to a host, and then disconnecting from it and checking the connection.

Pseudocode

Create host
connectToHost(host)
disconnectFromHost()
assert(not_connected)

dropConnections: The drop connections function is called when a host leaves an active match. This will gracefully stop the program form crashing on every player's machine. It will be tested by creating a match and having 2 players join the host. The host will leave the match and we will check that all player connections were dropped safely.

Pseudocode

Create host
Create 2 players
connectToHost(players)
host.dropConnections()
assert(players are no longer connected

openHostConnection: The open host connection function is called when a user decides to host a game. This will open connection to the host for other players. We will test this by opening the connection and checking that it is joinable.

Pseudocode

openHostConnection
assert(can_connect)

4.2.2 View

The view will be made up of purely visual components, and thus will not be tested by automation. We will instead test the view and its classes manually meaning these tests will be simple UI tests to ensure buttons are clickable and graphics display properly, which will be determined by the team at the time of testing.

4.2.3 Controller

The Game class acts as the controller for our system. It is responsible for communicating with the model and view to ensure the system runs smoothly. The game contains functions that are called by the view when a user interacts with it, and functions that are called to communicate with the model to get updates on the game state. The game class will handle communications with the RobotLibrarian and the Server, to get the external info it needs to begin the game. After a game has started, the game class will continually run to move the game forward.

The game class is the point in the system that all commands and data pass through, and thus it is crucial to thoroughly test this module. This class will run the main program, and coordinate all game data. It is important that the game can effectively communicate with both the view and the model to ensure all modules receive the information they need to progress the program.

We will create a J-unit test module for the controller class that will contain methods for testing the functions in the game class as described below.

gameInit: The gameInit() function will initialize all players and their robots. This function will be tested by calling it with a few players and testing to see if the players were properly initialized. A few different, but non-exhaustive tests will be sufficient. We will initialize a game with two human players and check to see that we have a list of two players with all their robots initialized. Then a game with three AI players will be initialized and checked in the same way. Finally, a game with 6 players, 3 human and 3 AI will be initialized and checked.

Pseudocode:

create 2 fake human players
gameInit()
assert(2_players_in_player_list)
assert(each_player_has_a_colour)

assert(each_player_has_three_robots)

create 3 fake AI players
gameInit()
assert(3_players_in_player_list)
assert(each_player_has_a_colour)
assert(each_player_has_three_robots)

create 3 fake human players
create 3 fake AI players
gameInit()
assert(6_players_in_player_list)
assert(each_player_has_a_colour)
assert(each_player_has_three_robots)

joinGame: The join game function is called by the StartView. It will specify a host to connect to, and will call the connectToHost() function in the server. This will be tested by creating a fake host to connect to, and calling the join game with that host. The parameters will be checked for correctness.

Pseudocode:

create a fake host
joinGame(fake_host)
server.connectToHost(fake_host)
assert(user_is_connected_to_host)

hostGame: The host game function is called by the StartView. It will open a host to be connected to by other users. It will call the openHostConnection function in the server. This function will be tested by creating a host and testing to see if the connection was properly opened.

Pseudocode:

hostGame()
assert(connection_is_open)

lobbyBack: The lobby back function is called by the LobbyView. It will take the user back to the StartView and properly disconnect the user from the host. If the user who selected the back option is the host of the game, the match is terminated and all players are kicked. This will be tested by connecting the user to a fake host, and calling the appropriate server functions to terminate the connection.

Pseudocode:

server.connect(fake_host)
lobbyBack()

assert(user_is_not_connected)

server.host(me)
lobbyBack()
assert(match_does_not_exist)

switch: The switch function is called by the LobbyView. If called by a player, it changes the player to an observer. If called by an observer, it changes the observer to a player. This function will be tested by creating a player object and switching them to an observer, then switching the new observer back to a player.

Pseudocode:

create player
switch(player)
assert(the_user_is_now_an_observer)
switch(observer)
assert(the_user_is_now_a_player)

kick: The kick function is called by the LobbyView. It is used to kick a player from the game. When called, the chosen player is disconnected from the host. To test this we will connect a player to a host, and kick the player. We will then check to see if the player is still connected.

Pseudocode:

create host
create player
connect player to host
kick(player)
assert(player_is_not_connected_to_host)

tileClicked: The tile clicked function is called by the GameView. When it is called, the tile object located at the coordinates of the mouse is returned. This function will be tested by selecting a tile in game and checking if the selected tile is equal to the tile returned by the tile tileClicke() function.

Pseudocode:

gameInit()
player1 clicks on Tile
assert(selected_tile == tileClicked())

endTurn: The end turn function is called by the GameView. This signals that a player is finished their turn and the next player should be notified that it is their turn. This

can be tested by initializing a game with two players. The end turn function is called by one player, and we will check if the next player is given their turn.

Pseudocode:

create 2 player match
endTurn(player1)
assert(player2_is_turn)

checkWinLose: The check win lose function is called by the GameView. It will be called at the end of each round, to determine if a team has won. This will be tested by creating a game with two players. The second player's robots will be destroyed, meaning that the first player wins. We will check this case.

Pseudocode:

create 2 player match
player 2 robots.setHealth(0)
checkWinLose()
assert(player1_wins)

getTileInfo: This function is called by the GameView when the inspect method is called. The given tile will be passed to the model, and a list of all robots will be returned. This will be tested by creating a tile with 3 arbitrary robots on it. The method will be called on this tile and verified that the correct info is returned.

Pseudocode:

create a tile
put 3 robots on it
getTileInfo(theTile)
assert(tile_info_matches_input)

moveRobot: The move robot function is called by the GameView when the move method is called. The given robot is moved to the given tile. Because the tiles are checked for movability by the tileClcked() function, there should never be a tile that cannot be moved to. The test for this function will be to put a robot on a tile, and then move them to another tile. Then the robots current tile will be checked to ensure it is on the correct tile.

Pseudocode:

create an array of tiles
put a robot on a tile
moveRobot(a_new_tile)
assert(robot_is_on_a_new_tile)

attackTile: The attack tile function is called by the GameView when the attack method is called. The given robot attacks the given tile, and all robots on that tile are

damaged by the robot's attack points. To test this, we will create some robots and an array of tiles. We will place the robots on some of the tiles. Then one new robot is created and placed in range of all robots. Each tile will be attacked, and the robots on the tile will be checked to see if they took the correct damage. Tiles without any robots will also be attacked, meaning no damage is taken.

Pseudocode:

```
create an array of tiles
place robots on the tiles
create a new robot
for each tile
        attackTile(new_robot, tile)
        assert(robots_on_tile_took_correct_damage)
end
```

updateRobotPrograms: The updateRobotPrograms function is called at the beginning of a game. This will call methods in the robot librarian to collect the necessary robot programs. This will be tested by calling the function to retrieve some robot programs. We will check that the correct programs were retrieved.

Pseudocode:

```
updateRobotPrograms()
assert(programs_downloaded)
```

checkServerStatus: The check server status function is called at the beginning of each turn. It will return an error if the connection has been lost. This will be tested by creating a connection, and calling the function. It should return with no error. We will then drop the connection, and test it again. This should return an error.

Pseudocode:

```
create server
checkServerStatus()
assert(connection_good)
disconnect
checkServerStatus()
assert(error)
```

sendGameStateToServer: The sendGameStateToServer function is called at the end of each player's turn. It will send the current game state to the server to be distributed to all players. We will test this by creating a game state and sending it to another game. If the game state is recieved successfully, the test is passed.

Pseudocode:

```
create server
create game state
sendGameStateToServer()
assert(game_state_successfully_recieved)
```

5 Summary

We have outlined our approach to testing the Robot Game system. We will be using J-Unit testing to write our testing modules, which will allow us to automate a large portion of our tests. The view level of the system will be tested manually by users. Overall we believe we have covered in sufficient detail the test cases required to ensure our system runs smoothly.