**War of Robotcraft**

**Test Plan**

Team: A3

Team Members:

|  |  |  |
| --- | --- | --- |
| **Name** | **NSID** | **Student ID** |
| Fu, Chen | chf354 | 11183491 |
| He, Jiahuan | jih889 | 11183346 |
| Wang, Shisong | shw940 | 11157916 |
| Xie, Ruida | rux793 | 11194258 |
| Yang, Chen | chy202 | 11183550 |

Date: Nov 01, 2016

|  |  |  |
| --- | --- | --- |
| **Version Number** | **Description of Changes** | **Approved Date** |
| 0.1 | First Draft | 2016-11-01 |
| 0.2 |  |  |
| 0.3 |  |  |
| 0.4 |  |  |
| 0.5 |  |  |
| 0.6 |  |  |
| 0.7 |  |  |
| 1.0 |  |  |

Revision History:

# Introduction

The test plan document demonstrates testing methods and procedures to the War of Robotcraft project team. The document provides each team member a guide of writing testing code by generating sufficient test cases and sequences in Pseudocode form.

* 1. **Objectives**

First, all the methods of each class are to be tested to ensure that all methods can work properly as independent units. Next, several methods can achieve collaboration with others from different architecture components in order to perform some sequential actions. In this step, the communications between the components of the architecture are emphasized. Finally, we are aiming for integrating all the functionalities into an entire system.

* 1. **Approaches**

We divide the system into different interfaces to test based on the collaborative relationships (Detailed information is introduced below). For each interface, we’ve designed unit tests for all the methods of each related class for the purpose of ensuring each method work correctly. In addition, functional tests examine whether a group of methods can work together to achieve some sequences of functionalities. Integration test is provided after ensuring that all individual components can achieve their own functionalities.

* 1. **Interfaces**

Since the number of interfaces to be tested is determined by the architecture of the software system. Our War of Robotcraft uses MVC architecture, hence we group the interfaces by considering what classes cross MVC components have to work together in order to perform some actions, and each interface is responsible for a specific series of functionalities. In this case, there are four interfaces in our system. The first one is “Initialize game interface”, which is responsible for executing a series of actions to initialize the game. The actions are displaying “SetGameModeView”, “GameStartView”, “GarageView” and “GameBoardView”. Along with these actions, user’s input data and the data fetched from robot librarian is passed through the controller to model classes. The second interface “Human Player Interface” has functionalities of all players’ and robots’ actions like move or shoot. All three components of the MVC architecture possess classes which are necessary to perform these actions. For example, if a player wants to move a robot, the controller have to fetch position and robot information from model classes and pass it to view class to display. Similarly, AI player has those actions as well. But the AI player has additional features such as mailbox and interpreter so that the collaborative procedure of AI player is slightly different from the human player, that is the reason why we separate “AI player interface” from “Human player interface”. Another interface is “Game interface”. This one is to keep the game running and updating the map while the game is in progress. An example of the data flow in this interface would be controller obtains player’s data, such as name, robots’ health points, and pass it to game map to display. In this test plan, each interpreter are tested separately and then follows a integration test.

# Initialize game interface

### Start Function

* Test case 1: test start in human vs human mode and three players
  + Test steps:
    1. Press start button.
    2. Select human vs human.
    3. Select three players
    4. Select player’s name
    5. Press confirm
  + Expected output:
    1. Game board initialized with three human players successful.
* Test case 2: test start in AI vs AI mode and two players
  + Test steps:
    1. Press start button.
    2. Select AI vs AI.
    3. Select two players
    4. Select player’s name
    5. Press confirm button
  + Expected output:
    1. Game board initialized with two AI players successful.
* Test case 3: test start in AI vs Human mode and six players
  + Test steps:
    1. Press start button.
    2. Select AI vs Human.
    3. Select six players
    4. Select player’s name
    5. Press confirm button
  + Expected output:
    1. Game board initialized with AI players and human players successful.

### Garage Function

* Test case 1: test revise
  + Test steps:
    1. Press garage button.
    2. Select the robot to be revised.
    3. Press revise button.
  + Expected output:
    1. Robot has been changed.
* Test case 2: test retire
  + Test steps:
    1. Press garage button.
    2. Select the robot to be retire
    3. Press retire button
  + Expected output:
    1. Robot has been retired.
* Test case 3: test register
  + Test steps:
    1. Press garage button.
    2. Add the robot to the garage.
    3. Press register button.
  + Expected output:
    1. A new robot has been register.

# Human player interface

## Unit tests

### Class Robot

#### isDead(): bool

Summary: This unit test is to test Robot.isDead() whether it can return true if a robot is dead or false otherwise.

robot 🡨 new Robot()

declare expectedValue

declare actualValue

* test case 1: the robot is not dead

expectedValue 🡨 false

robot.healthPoint 🡨 1

actualValue 🡨 robot.isDead()

assert if actualValue and excpectdValue are equal

* test case 2: the robot is dead

expectedValue 🡨 true

robot.healthPoint 🡨 0

actualValue 🡨 robot.isDead()

assert if actualValue and excpectdValue are equal

* test case 3: the value is out of range

expectedValue 🡨 true

robot.healthPoint 🡨 -1

actualValue 🡨 robot.isDead()

assert if actualValue and excpectdValue are equal

#### turn()

Summary: This unit test is to test whether the robot can turn to the direction as the player want.

robot 🡨 new Robot()

* Test case 1: the robot is turn to the direction 0, which does not turn.

expectedDirection = 0

robot.direction = 0

robot.turn(expectedDirection)

actualDirection = robot.direction

assert if expectedDirection and actualDirection are equal

* Test case 2: the robot is turn to the direction 5, which is the furthest position.

expectedDirection = 5

robot.direction = 0

robot.turn(expectedDirection)

actualDirection = robot.direction

assert if expectedDirection and actualDirection are equal

* Test case 3: the robot is turn to the direction -1, which is the incorrect input less than 0.

expectedDirection = -1

try robot.turn(expectedDirection)

catch parameter out of bound exception

* Test case 4: the robot is turn to the direction 6, which is the incorrect input larger than 5.

expectedDirection = 6

try robot.turn(expectedDirection)

catch parameter out of bound exception

#### move()

Summary: This unit test is to test Robot.move() whether it can change the robot position correctly.

robot 🡨 new Robot()

robot.coor.x 🡨 4

robot.coor.y 🡨 -4

robot.coor.z 🡨 0

coor 🡨 new coordinate()

coor.x 🡨 3

coor.y 🡨 -3

coor.z 🡨 0

declare excpectedValueMovePoint

declare expectedValueCoor

declare actualValueMovePoint

declare actualValueCoor

* test case 1: the robot has full of movementPoint, then movemetPoint minus 1, the coor will changed at new position

robot.movementPoint 🡨 3

robot.move(coor)

excpectedValueMovePoint 🡨 2

actualValueMovePoint 🡨 robot.movementPoint

assert if actualValueMovePoint and excpectedValueMovePoint are equal

expectedValueCoor 🡨 coor

actualValueCoor 🡨 robot.coor

assert if actualValueCoor and expectedValueCoor are equal

* test case 2: the robot has no movementPoint, then robot cannot move, nothing to be changed

robot.movementPoint 🡨 0

robot.move(coor)

excpectedValueMovePoint 🡨 0

actualValueMovePoint 🡨 robot.movementPoint

assert if actualValueMovePoint and excpectedValueMovePoint are equal

expectedValueCoor 🡨 robot.coor

actualValueCoor 🡨 robot.coor

assert if actualValueCoor and expectedValueCoor are equal

#### shoot(int distance): Coordinate coor

Summary: This unit test is to test Robot.shoot() which receives a distance to shoot and returns the target coordinate.

robot 🡨 new Robot()

robot.coordinate.x 🡨 0

robot.coordinate.y 🡨 0

robot.coordinate.z 🡨 0

robot.direction 🡨 2

robot.hasShot 🡨 false

declare expectedCoordinate

declare actualCoordinate

declare expectedHasShot

declare actualHasShot

* test case 1: The robot shoots when hasShot is false and the distance is 0

expectedCoordinate 🡨 new Coordinate()

expectedCoordinate.x 🡨 0

expectedCoordinate.y 🡨 0

expectedCoordinate.z 🡨 0

robot.shoot(0)

actualCoordinate 🡨 robot.coor

assert if actualCoordinate and expectedCoordinate are equal

expectedHasShot 🡨true

actualHasShot 🡨 robot.hasShot

assert if actualHasShot and expectedHasShot are equal

* test case 2: The robot shoots when hasShot is false and the distance is between 0 and the maximum

expectedCoordinate 🡨 new Coordinate()

expectedCoordinate.x 🡨 -2

expectedCoordinate.y 🡨 0

expectedCoordinate.z 🡨 2

robot.hasShot 🡨 false

robot.shoot(2)

actualCoordinate 🡨 robot.coor

assert if actualCoordinate and expectedCoordinate are equal

expectedHasShot 🡨true

actualHasShot 🡨 robot.hasShot

assert if actualHasShot and expectedHasShot are equal

* test case 3: The robot shoots even when hasShot is false but the distance is out of range

robot.hasShot 🡨 false

try robot.shoot()

check if catch an exception

* test case 4: The robot shoots when hasShot is true

try robot.shoot()

check if catch an exception

#### demaged()

Summary: This unit test is to test Robot.damaged(int attackPoint) it it can change robots’ health point correctly.

robot 🡨 new Robot()

declare expectedValue

declare actualValue

* test case 1: the robot is not dead

robot.healthPoint = 3

expectedValue = 1

robot.healthPoint = 1

actualValue = robot.damaged(2)

assert if actualValue and excpectdValue are equal

* test case 2: the robot is dead

expectedValue = 0

robot.healthPoint = 1

actualValue = robot.damaged(2)

assert if actualValue less than or equal to excpectdValue

### Class Player

#### isDead(): bool

Summary: This unit test is to test Player.isDead() whether it can return true if a player is dead or false otherwise.

player 🡨 new Player()

declare expectedValue

declare actualValue

* test case 1: the player is not dead when all robot alive

expectedValue 🡨 false

player.scoutRobot.healthPoint 🡨 1

player.sniperRobot.healthPoint 🡨 2

player.tankRobot.healthPoin 🡨 1

actualValue 🡨 player.isDead()

assert if actualValue and excpectdValue are equal

* test case 2: the player is dead when all robot are dead

expectedValue 🡨 true

player.scoutRobot.healthPoint 🡨 0

player.sniperRobot.healthPoint 🡨 0

player.tankRobot.healthPoin 🡨 0

actualValue 🡨 player.isDead()

assert if actualValue and excpectdValue are equal

* test case 2: the player is dead when only one robot alive

expectedValue 🡨 false

player.scoutRobot.healthPoint 🡨 1

player.sniperRobot.healthPoint 🡨 0

player.tankRobot.healthPoin 🡨 0

actualValue 🡨 player.isDead()

assert if actualValue and excpectdValue are equal

#### getCurrentRobot(): Robot

Summary: this unit test is to test whether the player can get the correct robot on the current play, i.e. the alive robot has the highest movement point.

humanPlayer 🡨 new humanPlayer()

declare expectRobot

declare actualRobot

* test case 1: all robots alive, in the first play of a player, the current robot should be Scout.

humanPlayer.scoutRobot.hasMoved = false

humanPlayer.sniperRobot.hasMoved = false

humanPlayer.tankRobot.hasMoved = false

expectRobot.type = scout

actualRobot = humanPlayer.getCurrentRobot()

assert if expectRobot.type and actualRobot.type are equal

* test case 2: all robots alive, in the second play of a player, the current robot should be Sniper.

humanPlayer.scoutRobot.hasMoved = true

humanPlayer.sniperRobot.hasMoved = false

humanPlayer.tankRobot.hasMoved = false

expectRobot.type = sniper

actualRobot = humanPlayer.getCurrentRobot()

assert if expectRobot.type and actualRobot.type are equal

* test case 3: all robots alive, in the third play of a player, the current robot should be Tank.

humanPlayer.scoutRobot.hasMoved = true

humanPlayer.sniperRobot.hasMoved = true

humanPlayer.tankRobot.hasMoved = false

expectRobot.type = tank

actualRobot = humanPlayer.getCurrentRobot()

assert if expectRobot.type and actualRobot.type are equal

* test case 4: one robot died, the current player should be an alive robot who has the highest move point and false value in its hasMoved field.

humanPlayer.scoutRobot.healthPoint = 0

humanPlayer.sniperRobot.hasMoved = false

humanPlayer.tankRobot.hasMoved = false

expectRobot.type = sniper

actualRobot = humanPlayer.getCurrentRobot()

assert if expectRobot.type and actualRobot.type are equal

* Test case 5: one robot died, the current player should be an alive robot who has the highest move point and false value in its hasMoved field.

humanPlayer.sniperRobot.healthPoint = 0

humanPlayer.scoutRobot.hasMoved = false

humanPlayer.tankRobot.hasMoved = false

expectRobot.type = scout

actualRobot = humanPlayer.getCurrentRobot()

assert if expectRobot.type and actualRobot.type are equal

* Test case 6: one robot died, the current player should be an alive robot who has the highest move point and false value in its hasMoved field.

humanPlayer.sniperRobot.healthPoint = 0

humanPlayer.scoutRobot.hasMoved = true

humanPlayer.tankRobot.hasMoved = false

expectRobot.type = tank

actualRobot = humanPlayer.getCurrentRobot()

assert if expectRobot.type and actualRobot.type are equal

* Test case 7: two robots died, the current player should be the alive robot.

humanPlayer.sniperRobot.healthPoint = 0

humanPlayer.scouptRobot.healthPoint = 0

humanPlayer.tankRobot.hasMoved = false

expectRobot.type = tank

actualRobot = humanPlayer.getCurrentRobot()

assert if expectRobot.type and actualRobot.type are equal

* Test case 8: two robots died, the current player should be the alive robot.

humanPlayer.scouptRobot.healthPoint = 0

humanPlayer.tankRobot.healthPoint = 0

humanPlayer.sniperRobot.hasMoved = false

expectRobot.type = sniper

actualRobot = humanPlayer.getCurrentRobot()

assert if expectRobot.type and actualRobot.type are equal

* Test case 9: all robots died, the current player should be none. This player will lose the game.

humanPlayer.scouptRobot.healthPoint = 0

humanPlayer.sniperRobot.healthPoint = 0

humanPlayer.tankRobot.healthPoint = 0

expectRobot = null

actualRobot = humanPlayer.getCurrentRobot()

assert if expectRobot and actualRobot are equal

#### goNextRobot(): void

Summary: this unit test is to test whether the player can go to perform the next robot, or go back to the first robot if current robot is the last.

humanPlayer 🡨 new humanPlayer()

declare expectRobot

declare actualRobot

* test case 1: the current robot is the first one, after going to the next robot, it is the second robot.

currentRobot = the largest movement point alive robot

expectRobot = the second largest movement point alive robot

humanPlayer.goNextRobot()

actualRobot = currentRobot

assert if expectRobot and actualRobot are equal

* test case 2: the current robot is the second one, after going to the next robot, it is the last robot.

currentRobot = the second largest movement point alive robot

expectRobot = the lowest movement point alive robot

humanPlayer.goNextRobot()

actualRobot = currentRobot

assert if expectRobot and actualRobot are equal

* test case 3: the current robot is the last one, after going to the next robot, it is the first robot.

currentRobot = the lowest movement point alive robot

expectRobot = the largest movement point alive robot

humanPlayer.goNextRobot()

actualRobot = currentRobot

assert if expectRobot and actualRobot are equal

### Human Player

#### Move(): void

Summary: This unit test is to test whether a player can change its robot’s position correctly.

Player🡨 new HumanPlayer()

player.getCurrentRobot().coor.x 🡨 4

player.getCurrentRobot().coor.y 🡨 -4

player.getCurrentRobot().coor.z 🡨 0

coor 🡨 new coordinate()

coor.x 🡨 3

coor.y 🡨 -3

coor.z 🡨 0

declare excpectedValueMovePoint

declare expectedValueCoor

declare actualValueMovePoint

declare actualValueCoor

* test case 1: the player’s robot has full of movementPoint, then movemetPoint minus 1, the coor will changed to a new position

player.getCurrentRobot().movementPoint 🡨 3

player.getCurrentRobot().move(coor)

excpectedValueMovePoint 🡨 2

actualValueMovePoint 🡨 player.getCurrentRobot().movementPoint

assert if actualValueMovePoint and excpectedValueMovePoint are equal

expectedValueCoor 🡨 coor

actualValueCoor 🡨 player.getCurrentRobot().coor

assert if actualValueCoor and expectedValueCoor are equal

* test case 2: the player’s robot has no movementPoint, then player’s robot cannot move, nothing to be changed

player.getCurrentRobot().movementPoint 🡨 0

player.getCurrentRobot().move(coor)

excpectedValueMovePoint 🡨 0

actualValueMovePoint 🡨 player.getCurrentRobot().movementPoint

assert if actualValueMovePoint and excpectedValueMovePoint are equal

expectedValueCoor 🡨 player.getCurrentRobot().coor

actualValueCoor 🡨 player.getCurrentRobot().coor

assert if actualValueCoor and expectedValueCoor are equal

#### Turn(int direction): void

Summary: This unit test is to test whether a player can operate a robot to turn.

player🡨 new HumanPlayer()

* Test case 1: the player operates its robot turn to the direction 0, which does not turn.

expectedDirection = 0

player.getCurrentRobot().direction = 0

player.getCurrentRobot().turn(expectedDirection)

actualDirection = player.getCurrentRobot().direction

assert if expectedDirection and actualDirection are equal

* Test case 2: the player operates its robot turn to the direction 5, which is the furthest position.

expectedDirection = 5

player.getCurrentRobot().direction = 0

player.getCurrentRobot().turn(expectedDirection)

actualDirection = player.getCurrentRobot().direction

assert if expectedDirection and actualDirection are equal

* Test case 3: the player operates its robot turn to the direction -1, which is the incorrect input less than 0.

expectedDirection = -1

try player.getCurrentRobot().turn(expectedDirection)

catch parameter out of bound exception

* Test case 4: the robot is turn to the direction 6, which is the incorrect input larger than 5.

expectedDirection = 6

try player.getCurrentRobot().turn(expectedDirection)

catch parameter out of bound exception

#### Shoot(Coordinate coor): void

Summary: This unit test is to test if a player can use its robots to shoot.

player.getCurrentRobot()🡨 new Robot()

player.getCurrentRobot().coordinate.x 🡨 0

player.getCurrentRobot().coordinate.y 🡨 0

player.getCurrentRobot().coordinate.z 🡨 0

player.getCurrentRobot().direction 🡨 2

player.getCurrentRobot().hasShot 🡨 false

declare expectedCoordinate

declare actualCoordinate

declare expectedHasShot

declare actualHasShot

* test case 1: The player operates its robot shoot when hasShot is false and the distance is 0

expectedCoordinate 🡨 new Coordinate()

expectedCoordinate.x 🡨 0

expectedCoordinate.y 🡨 0

expectedCoordinate.z 🡨 0

robot.shoot(0)

actualCoordinate 🡨 robot.coor

assert if actualCoordinate and expectedCoordinate are equal

expectedHasShot 🡨true

actualHasShot 🡨 robot.hasShot

assert if actualHasShot and expectedHasShot are equal

* test case 2: The player operates its robot shoot when hasShot is false and the distance is between 0 and the maximum

expectedCoordinate 🡨 new Coordinate()

expectedCoordinate.x 🡨 -2

expectedCoordinate.y 🡨 0

expectedCoordinate.z 🡨 2

robot.hasShot 🡨 false

robot.shoot(2)

actualCoordinate 🡨 robot.coor

assert if actualCoordinate and expectedCoordinate are equal

expectedHasShot 🡨true

actualHasShot 🡨 robot.hasShot

assert if actualHasShot and expectedHasShot are equal

* test case 3: The player operates its robot shoot even when hasShot is false but the distance is out of range

robot.hasShot 🡨 false

try robot.shoot()

check if catch an exception

* test case 4: The player operates its robot shoot when hasShot is true

try robot.shoot()

check if catch an exception

## Functional tests

### Human Player Turning Function

* Test case 1: choosing direction 0, no turn
  + Test steps:
    1. Press key “T” to enter turning mode.
    2. Press key “0” to choose direction 0.
  + Expected output:
    1. Nothing should happen.
* Test case 2: choosing direction from 1 to 5, turn to the selected direction
  + Test steps:
    1. Press key “T” to enter turning mode.
    2. Press key “3” to choose direction 3.
  + Expected output:
    1. The robot should have turned 180 degrees.
    2. The facing direction is new direction 0.
* Test case 3: choosing direction out of the range, no turn
  + Test steps:
    1. Press key “T” to enter turning mode.
    2. Press key “8” as a wrong input.
  + Expected output:
    1. Nothing should happen.

### Human Player Moving Function

* Test case 1: no moving point, no move
  + Test steps:
    1. Press key “M” to move when having no moving point.
  + Expected output:
    1. Nothing should happen.
* Test case 2: having moving point but closing to border, no move
  + Test steps:
    1. Press key “M” to move when on to the edge of the game board.
  + Expected output:
    1. Nothing should happen.
* Test case 3: having moving point and not closing to border, no move
  + Test steps:
    1. Press key “M” to move.
  + Expected output:
    1. The robot on the play has moved 1 space along the direction it faces.

### Human Player Shooting Function

* Test case 1: shooting to an open area
  + Test steps:
    1. Press key “S” to enter shooting mode.
    2. Press a number for the distance, which targeting an area without any robot.
  + Expected output:
    1. The operating robot is marked as “has shot”.
    2. Nothing else should happen.
* Test case 2: shooting to a robot which has health point lower than the attack point of the operating robot
  + Test steps:
    1. When operating a tank, close to an enemy scout.
    2. Turn to the enemy scout.
    3. Press key “S” to enter shooting mode.
    4. Press number key to target the enemy scout.
  + Expected output:
    1. The enemy scout is destroyed.
    2. The operating robot is marked as “has shot”.
* Test case 3: shooting to a robot which has health point higher than the attack point of the operating robot
  + Test steps:
    1. When operating a scout, close to an enemy tank.
    2. Turn to the enemy tank.
    3. Press key “S” to enter shooting mode.
    4. Press number key to target the enemy tank.
  + Expected output:
    1. The enemy tank is damaged.
    2. The operating robot is marked as “has shot”.
* Test case 4: shooting to an area that has multiple robots
  + Test steps:
    1. Move to somewhere near an area with more than one robot.
    2. Turn to that direction.
    3. Press key “S” to enter shooting mode.
    4. Press number key to target those robots.
  + Expected output:
    1. The robots in the targeted area are damaged, and if the health points of targeted robots are lower than the attack point of the operating robot, the targeted robots are destroyed.
    2. The operating robot is marked as “has shot”.
* Test case 5: shooting with 0 distance
  + Test steps:
    1. Press “S” to enter shooting mode.
    2. Press “0” to shoot in situ.
  + Expected output:
    1. The operating robot is damaged. If the robot’s health points are lower than attack point, then it would be destroyed.

# AI player interface

# Game interface

## Unit Tests

### Class Game

#### setPlayerPositions():void

Summary: This unit test is to test Game.setPlayerPositions() which sets the player positions into player list and set the player at the first position as the currentPlayer.

game 🡨 new Game()

players 🡨 new Player[6]

players[0] 🡨 new Player()

players[1] 🡨 new Player()

players[2] 🡨 new Player()

players[3] 🡨 new Player()

players[4] 🡨 new Player()

players[5] 🡨 new Player()

declare expectedValue

declare actualValue

expectedPlayer 🡨 players[0]

declare actualPlayer

* test case 1: there are two players

for i from 0 to 1

game.playerList.insert(new Pair<int, Player>(0, players[i]))

done

game.setPlayerPositions()

expectedValue 🡨 0

actualValue 🡨 game.playerList.indexOf(0).indexOf(0)

assert if actualValue and excpectdValue are equal

expectedValue 🡨 3

actualValue 🡨 game.playerList.indexOf(1).indexOf(0)

assert if actualValue and excpectdValue are equal

actualPlayer 🡨 game.currentPlayer

assert if actualPlayer and expectedPlayer are equal

* test case 2: there are three players

game.playerList.insert(new Pair<int, Player>(0, players[2]))

game.setPlayerPositions()

expectedValue 🡨 0

actualValue 🡨 game.playerList.indexOf(0).indexOf(0)

assert if actualValue and excpectdValue are equal

expectedValue 🡨 2

actualValue 🡨 game.playerList.indexOf(1).indexOf(0)

assert if actualValue and excpectdValue are equal

expectedValue 🡨 4

actualValue 🡨 game.playerList.indexOf(2).indexOf(0)

assert if actualValue and excpectdValue are equal

actualPlayer 🡨 game.currentPlayer

assert if actualPlayer and expectedPlayer are equal

* test case 3: there are not two, three, or six players

game.playerList.insert(new Pair<int, Player>(0, players[3]))

try game.setPlayerPositions()

check if catch an exception

* test case 4: there are six players

for i from 0 to 1

game.playerList.insert(new Pair<int, Player>(0, players[i + 3]))

done

game.setPlayerPositions()

expectedValue 🡨 0

actualValue 🡨 game.playerList.indexOf(0).indexOf(0)

assert if actualValue and excpectdValue are equal

expectedValue 🡨 1

actualValue 🡨 game.playerList.indexOf(1).indexOf(0)

assert if actualValue and excpectdValue are equal

expectedValue 🡨 2

actualValue 🡨 game.playerList.indexOf(2).indexOf(0)

assert if actualValue and excpectdValue are equal

expectedValue 🡨 3

actualValue 🡨 game.playerList.indexOf(3).indexOf(0)

assert if actualValue and excpectdValue are equal

expectedValue 🡨 4

actualValue 🡨 game.playerList.indexOf(4).indexOf(0)

assert if actualValue and excpectdValue are equal

expectedValue 🡨 5

actualValue 🡨 game.playerList.indexOf(5).indexOf(0)

assert if actualValue and excpectdValue are equal

actualPlayer 🡨 game.currentPlayer

assert if actualPlayer and expectedPlayer are equal

#### goNextPlayer():void

Summary: This unit test is to test Game.goNextPlayer() which returns a player that should play next.

game🡨 new Game()

players 🡨 new Player[2]

players[0] 🡨 new Player()

players[1] 🡨 new Player()

declare expectedValue

declare actualValue

* test case 1: the current player is the first one

expectedValue 🡨 players[1]

game.goNextPlayer()

actualValue 🡨 game.currentPlayer

assert if actualValue and excpectdValue are equal

* test case 2: the current player is the last one

expectedValue 🡨 players[0]

game.goNextPlayer()

actualValue 🡨 game.currentPlayer

assert if actualValue and excpectdValue are equal

#### runPlay():void

Summary: This unit test is to test the whether a play can run as the current player accesses the permit to run a new play.

game 🡨 new game()

declare map

declare playerList

declare currentPlayer

* Test case 1: the current player is dead, so the player can not start a new play.

Try curerntPlayer.runPlay()

Catch null exception

* Test case 2: the player is not dead, but only one robot alive.

expectedMap 🡨 game.map + the robot moves

currrentPlayer.currentRobot.move()

currrentPlayer.currentRobot.shoot()

game.updateGame()

assert if the expectedMap and game.map() are equal

* Test case 3: the player is alive with all robots alive.

expectedMap 🡨 game.map + the robot moves

currrentPlayer.currentRobot.move()

currrentPlayer.currentRobot.shoot()

game.updateGame()

expectedPlayer 🡨 the next player of game.currentPlayer

game.goNextPlayer()

game.runPlay()

assert if the expectedMap and game.map() are equal

assert if the game.currentPlayer and the expectedPlyaer are equal

#### updateGame():void

### Class Map

#### Map(int playerNum)

Summary: This unit test is to test Map.Map(int playerNum) which is the constructor of Map class.

declare expectedCount

declare actualCount

declare expectedSize

declare actualSize

* test case 1: the number of player is 2 or 3

map 🡨 new Map(2)

expectedCount 🡨 61

actualCount 🡨 map.coordinateList.size()

assert if actualCount and excpectdCount are equal

expectedSize 🡨 2

actualSize 🡨 map.mapSize

assert if actualSize and excpectdSize are equal

* test case 2: the number of player is 6

map 🡨 new Map(6)

expectedCount 🡨 127

actualCount 🡨 map.coordinateList.size()

assert if actualCount and excpectdCount are equal

expectedSize 🡨 6

actualSize 🡨 map.mapSize

assert if actualSize and excpectdSize are equal

* test case 3: if the number is not 2, 3, or 6

try map 🡨 new Map(6)

check if catch an exception

#### updateMist():void

## Functional Tests

# Integration Tests

## Human Player Interface – Game Interface

## AI Player Interface – Game Interface

## Login Interface – Game Interface

# Changes

* 1. **Amendment**

We find a problem of the design document. After the last submission, we realized that we missed the AI player class in our final version. The reason of that was a mistake occurred when using version control system. In detail, there was a version contains the AI player class, and then a new version which doesn’t contain the AI player class was committed to the repository and covered the previous version without solving conflict properly. From this issue, we’ve obtained some valuable experience about how to collaborate as a team member using source control system. First, when conflicts occur, the commit and push actions should be performed after discussion with other team members carefully. Then, before submit the final version, everyone should review the whole document thoroughly to detect potential problems.

* 1. **Shoot Method**

In the test plan, we revised the shoot() method in both Robot class and Player class since we realized that this method should return a coordinate to deliver a message of the destination information.

# Summary

This is the test plan document of the War of Robotcraft project. The first section is an introduction of the whole plan. It briefly describes the objectives and tasks of the plan and introduces our method of designing the four interfaces. Then, next few sections are the main body of this document, which explains how each interface is to be tested. Every interface section also provides pseudocode of detailed unit tests and sequential steps of functional tests. Within each unit test, there are sufficient cases cover upper boundary, lower boundary, normal case and invalid input in order to ensure the robustness of the system. In addition, the Changes section of the document describes the revisions in this document compared to the design document.