**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

# Initialize game interface

# Human player interface

## Unit test

### 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.

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

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

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

* test case 4: the player’s Scout robot died. In this play, the current robot should be Sniper.
* Test case 5: the player’s Snip

#### 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 test

### 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

# 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.

# Summary

Unit test

Class XXXX

method()

test case 1: balabalabala

input: balabalabala

expected output: balabalabala

test case 2:

test case ….

Functional test

Start game: actions and event