

War of Robotcraft

Deployment and Maintenance Document

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: Dec 01, 2016

Document History Log:

Version Number	Description of Changes	Approved Date
0.1	First Draft	2016-12-01
0.2	Add deployment document	2016-12-04
0.3	Add maintenance document	2016-12-04
0.4	Add standard delta	2016-12-07
0.5	Modified deployment document	2016-12-07
0.6	Modified maintenance document	2016-12-08
0.7	Modified standard delta	2016-12-09
1.0	Final edition.	2016-12-09

Table of Contents

Table of Contents	2
A. Deployment Document	4
1. Introduction	4
i. Target user	4
ii. Computer requirement	4
2. Game Rules	4
i. Basics	4
ii. Timer	4
iii. Rounds and Turns	4
iv. Move and shoot	4
v. Surrender	5
vi. How to win	5
3. Interfaces	5
i. Start the game.....	5
ii. Mode selection	6
iii. Game board	7
4. User Operations	8
i. Introduction	8
ii. Operations	8
5. User limitations.....	8
i. Status bar	8
ii. Regulation in non-network mode.....	8
B. Maintenance Document	9
1. An As-Build Architecture	9
2. Details of tricky/intricate/important bits of our system	10
3. How to compile and run your system.....	10
4. With the purpose of helping the next programmer in their task of maintaining or extending your systems	10
C. Standard Delta.....	11
1. Requirements were not met	11
i. Robot librarian	11
ii. Interpreter (large part of it).....	11

iii.	UI.....	11
2.	designs didn't hold up to construction	11
i.	UI.....	11
ii.	Model Package	12
iii.	View Package	12
iv.	Controller Package	13
v.	aiutil package	13
vi.	warofrobotcraft package	13
vii.	Resources package	14
3.	Bugs remain	14

A. Deployment Document

1. Introduction

Welcome to War of Robotcraft. This is a casual robot battle game with a neat interface and simple operations. The user can select three modes in this game, which are human player versus human player, human player versus AI player and AI player versus AI player. The user can choose the number of players from 2, 3 to 6. The map size of 2 and 3 players is a big hexagon with 5 hexagons for each side, and in the situation of 6 players, the map would be a hexagon with 7 hexagons composing each side.

i. Target user

The people who would like playing casual games.

ii. Computer requirement

The computer should have keyboard and mouse as input equipment. For software, the computer should be able to run Java Virtual Machine.

2. Game Rules

i. Basics

There are three modes of two, three or six players. Each player holds three robots, which are **scout**, **sniper** and **tank**. Each robot has 4 properties: **attack points**, **health points**, **movement points** and **range**. Because of the difference of robot properties, each robot has its pros and cons.

	Attack	Health	Movement	Range
Scout	1	1	3	2
Sniper	2	2	2	3
Tank	3	3	1	1

ii. Timer

We designed a time limitation for each human player. For each turn, each player has 20 seconds to finish all operations. If the time runs out, the current turn of the current player is terminated, and the game move on to next player starting a new turn.

iii. Rounds and Turns

One round consists of several turns. A turn's end means a player with at least one robot has played one robot unit. And a new turn is started by the next player. One round ends when all players have played all remaining robots once. Within each turn, each player performs the operations like shoot, move or turn directions on one robot, and each player has to finish within 20 seconds.

iv. Move and shoot

A robot can move at most the times of its movement points. And it can shoot at most once. It can turn its directions for any times. Movement is only towards the robot's current direction, and shoot is the same. If a player wants to move towards

a different direction, the player needs to turn into a different direction.

v. Surrender

If a human player decides not to finish the game and wants to quit the game earlier, the player can surrender in the middle of the game.

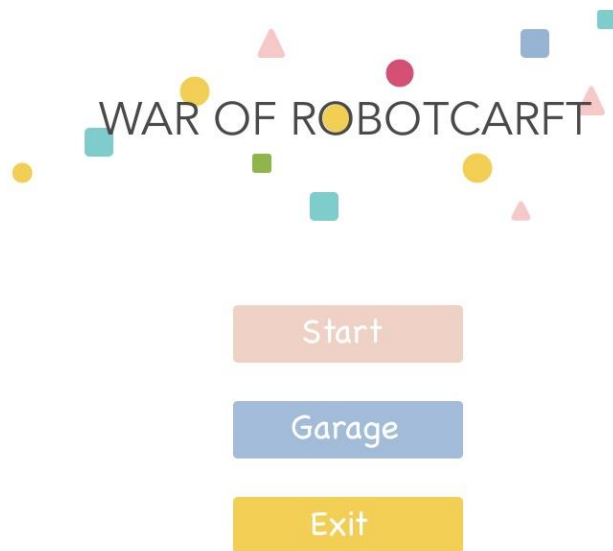
vi. How to win

Once there is only one player alive, the game ends. The remaining player is the winner. The winning situation can be formed from either all other players' robots are all dead or some players surrendered.

3. Interfaces

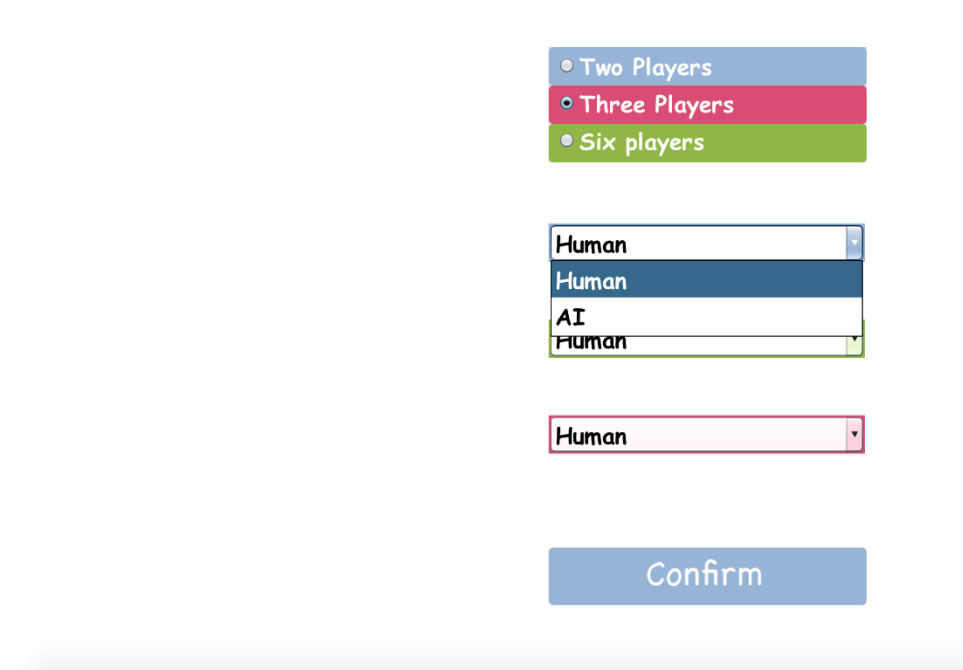
i. Start the game

When opening the game system, the game start interface is present on the screen as the following. There are three buttons you can click on **start**, **garage**, and **exit**. As the **start** button clicked, you will go to the next view where you can set up the mode for a new game. The **garage** button is to go the garage, but this view has not yet been done. The **exit** button is to exit the game system.



ii. **Mode selection**

This is the mode selection menu. As the game is designed, the user can choose a number of players from 2, 3 to 6. Once a user has selected the number of players, the corresponding number of selection boxes appear below. The user can select from Human and AI for each player. After the selection is finished, user confirm the selection by pressing the Confirm button below. Then the game board is opened.



The image shows a mode selection menu with three radio buttons for player count: 'Two Players' (blue), 'Three Players' (pink, selected), and 'Six players' (green). Below these are three dropdown menus for player type. The first dropdown is open, showing 'Human' (selected), 'Human', 'AI', and 'Human'. The second dropdown is closed and shows 'Human'. The third dropdown is closed and shows 'Human'. At the bottom is a blue 'Confirm' button.

Player Count	Player 1	Player 2	Player 3
Two Players			
Three Players	Human	Human	Human
Six players			

Confirm

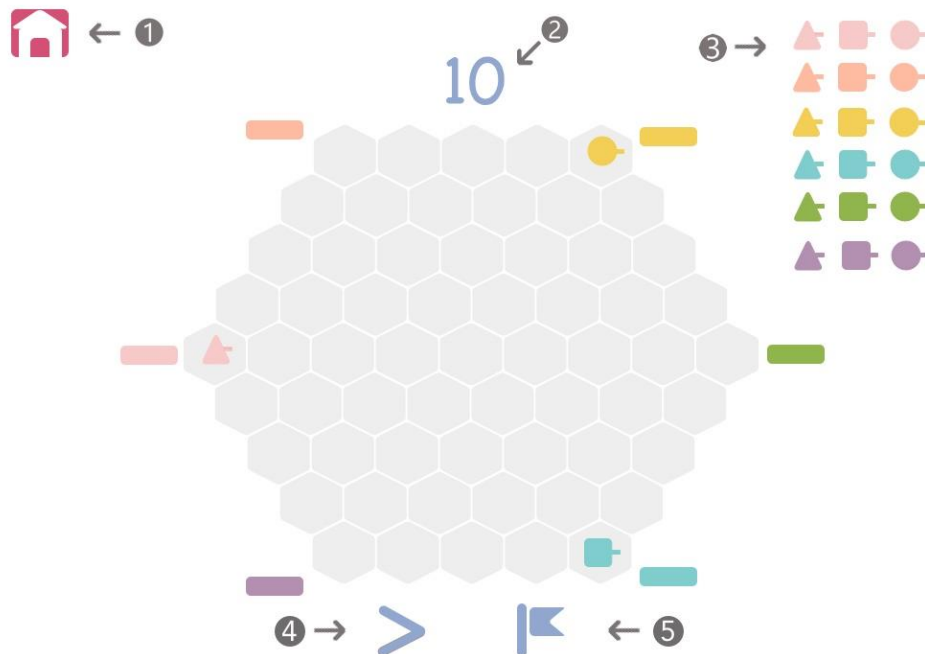
iii. Game board

a. Layout

This is the game board of robots. The hexagon in the center is the battlefield of the game. The six corners with colors are the positions where the robots are initialized. The pictures are showing the situation of three players. And for two players, the robots are initialized at the pink and green spots.

b. Buttons and indicators

The **button 1**, which is at the top left corner, is for returning to the main menu. The text pointed by **number 2** is a timer. Once the timer gets zero, the current turn is terminated, and the game goes to the next turn. The **number 3** at the top right corner is a status bar, which indicates current alive robot of the game. However, when we were implementing the game, we found that this status bar is not very useful for now, and the information given by the status bar will interfere the game's fairness, so we didn't implement the status bar. The detailed explanation is in below. The **button 4** provides the function of skipping the current turn if you don't feel like waiting for the timer ending. You can also press space to execute the same skipping current turn operation. The **button 5**, which is beside the skip button, is for surrendering when a player has decided to quit the game. If after one player surrendering there are still more than or equal to two players remaining, the game is still in progress, and the surrendered player is simply skipped.



4. User Operations

i. Introduction

The user operations are explained in this part. It is essential to be aware of these operations for performing a turn, move, shoot and skip current turn operations to play the game.

ii. Operations

Move is one of the basic operations, user press **m** for operating a robot to move one space forward in its current direction.

Turn is another basic operation, there are two steps to execute a turn. First, user press **t** to enter the turning mode. Then, user gives a direction to the robot by pressing a number. For example, if a user needs to turn right once, the user can press "**t**" first and "**1**" second. If a user needs to turn to the opposite direction, the user can press "**t**" first and "**3**" second.

Shoot operation is like turn, the user can press "**s**" first and then a number to choose the distance for shooting. For example, if a user wants to shoot at the spot two spaces ahead, the user can press "**s**" first and then press "**2**".

Finally, if a user needs to **skip the current turn** if the user finishes all the operations in the current turn and decides to carry on, the user can press "**Space**" to skip the current turn.

This operation can also be executed by click the **number 4 button** which is introduced in the previous part.

5. User limitations

i. Status bar

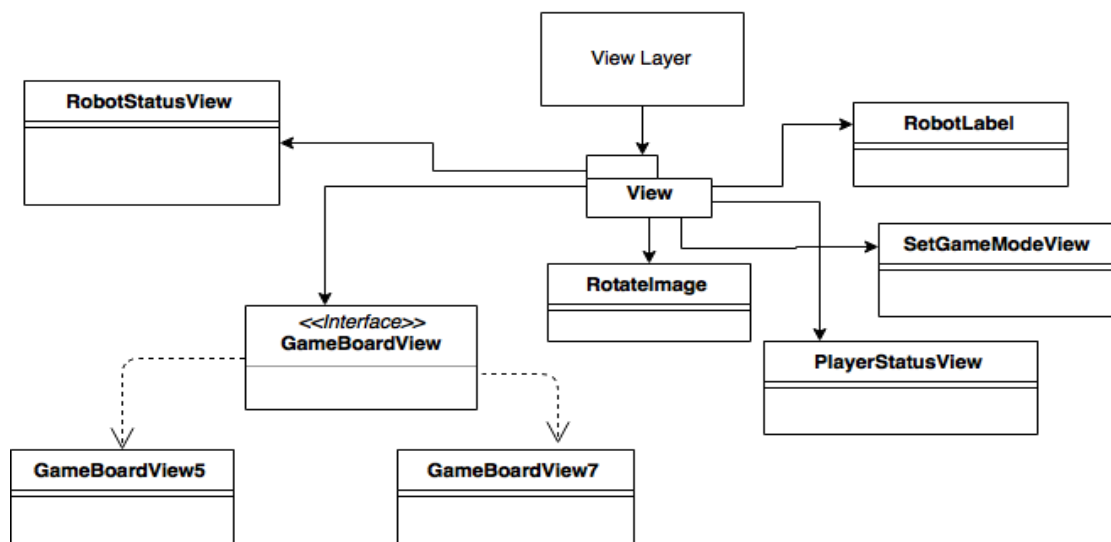
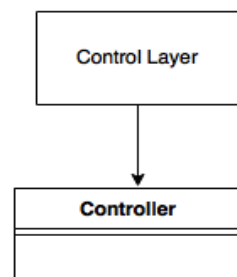
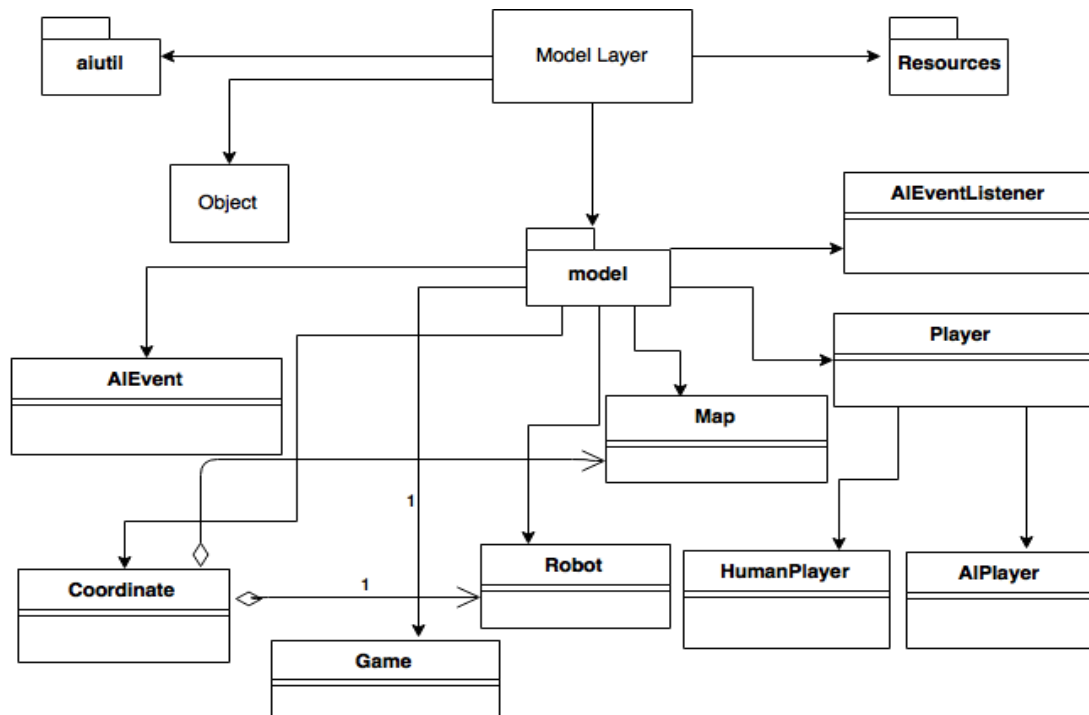
Initially, we designed a status bar to show the information of alive robots and every player can see it. But then we found it could interfere the fairness of the game, which is mentioned in the previous part of the document. The reason is that for the current user, it is no need for the user to know how many robots are possessed by other players because the area other than the user's visual range should be covered with war mist and should be invisible. Hence, the current player should only know how many robots the player has. So now the status bar is not functional on our game board.

ii. Regulation in non-network mode

Because we have not implemented the network part yet, it is a trick to prevent another player glance at the screen and remember the positions of the enemy's robots which should be invisible when the turn shifts. So there should be a rule for all players that they should not look at the screen when they end the current turn or while another player is playing the game. This is a limitation of our current vision of the game, and the problem will be solved if we can add the network mode onto the game.

B. Maintenance Document

1. An As-Build Architecture



2. Details of tricky/intricate/important bits of our system

Since the controller class communicates with classes in model layer and view layer, the controller class is the most important and intricate part. The controller class also have the responsibility of creating a new game and initial the game board and robot status. The controller receives all keyboard actions made by user and record AI operations to inform model layer to update robot status.

Another tricky part is the coordinate class of model package. The game board is a regular hexagon with side length of 5 or 7. It is hard to design a binary coordinate system for this game board so we designed a ternary coordinate system. The coordinate class is related to the gameBoardView class as well, each grid of the game board is named by its coordinate. Thus we can determine the position of each robot. Updating mist is relied on coordinate too because we need to calculate the explored area and different robot in one team can share their vision.

The runPlay() method in Game class is also intricate. This method should arrange update robot status. For example, is the robot moved? Shoot? This method should arrange each play and maintain the game running properly. The difference between this class and controller is, the controller is responsible for the logic operation, this class in Game package of Model layer is responsible for a data operation. This method can also find the winner player index and store it.

3. How to compile and run your system

First, clone the git remote repository to local repository. Then import the project from your local repository. At last, run the initial class, which is the main class of our system, then you can play the game as the user guide.

4. With the purpose of helping the next programmer in their task of maintaining or extending your systems

Our system has many areas where our projects can be expanded. First thing is we can add an online battle function, run this game on LAN using Socket. The second thing is we can add moving animation to AI robot, and shooting animation for all robots. The third area is we can implement the robot librarian function in the next stage, we tried it before but we did not figure it out due to the limitation of time. The most important thing is, we should extend the interpreter and forth script, in this version of our system, only part of forth words are supported by our interpreter.

C. Standard Delta

1. Requirements were not met

The following requirements were not met in the game:

i. Robot librarian

We have not yet done the part of robot librarian. That's mainly because we developed our own AI robots and use them to test and run, we put this requirement on lower level as we have other parts have to be done. However, we tried some external libraries, such as `httpcore`, `httpclient`, `common-io`, etc., but we just got the code 406 back and we have not figured out how to fix it.

ii. Interpreter (large part of it)

We have not yet completed all requirements for the interpreter, but it works for our own robots created by Forth script. Since the interpreter is a totally new thing to us, we take a lot of time to discuss and figure out how it works, how it combined to our code, what data structure we should use, and how to code it. At the end, we have done the interpreter to run our Forth scripts but not for any forth script.

iii. UI

We do not hold the design of the UI in some places, but we meet all requirements as the requirement document. See the modification details in the below.

2. designs didn't hold up to construction

We did a lot of modifications in the construction phase compared to the design. Basically, they are about the implementation of methods and classes. For the whole structure, we keep the design. As these modifications applied in our code, it not only brought us more productivity and efficiency but also improved the code reusability and time complexity.

i. UI

a. *Game Board View*

We changed the UI in the game board view where we play the game. The status diagram for each player was modified to a diagram with alive robots on the top right, which shows up the lives of robots for each player. That's because all robots will be dead in at most three shots, so this health point bar as shown in requirement document has less importance for the player. The alive robots present on the screen is enough to reminder the player.

b. *Game Mode Set View*

In this view, we delete the radio button to set the game mode in the choice of human vs human, human vs AI, and AI vs AI. As we constructing the game, we find that it useless for the player because, the player has to make choice of each player with the type of human or AI, which implies the mode.

ii. Model Package

a. *Coordinate Class*

We add `getRing()`, `getRange()` and `getNewCoordinate()` methods, and we overload the `gaRange` with different parameters. These methods all get a new coordinate or a range of coordinates based on a coordinate. They are helper functions dealing with robot move, turn, shoot and mist updating, which reduce our construction time and enhance code reusability.

b. *Game Class*

We add a field called `alivePlayerNumber`, this field stores the number of alive players. It helps the system determine who is the winner. We also split `updateGame()` method into 3 methods, `updateGameMove()`, `updateGameShootDamage()` and `updateGameShootDead()`, which makes the code more structural. This change separates the `updateGame()` method into pieces of smaller methods that handle many different situations including robot move, shoot, turn, damaged, do nothing, do nothing but move, surrender, etc. Thus, it enhances the reusability for other parts and the execution efficiency. Also, it facilitates us to debug in the construction phase.

c. *Map Class*

We use a hashmap to store coordinates and players' information instead of using linked list with the type of pair. When in the construction phase, we find it is difficult to write the code for the map. Then we discuss and change the data structure to hashmap because we believe it will bring us convenience in two aspects at least. The first one is easy to code which provide productivity in construction. The second is that it reduce time complexity relative to the linked list since there many retrieves and set back operations when the game running.

d. *Player Class*

We added a `currentRobot` field to store the robot for playing because it can help us to simplify the process to get an object of current playing robot. For the easy access to the view range of all robots in a team, we added a field called `viewRangeList` to store the view range of all robots in the same team and a method called `updateViewRange` to update the field. Also, we added a surrender method for the player to give up, which was not considered as part of our design, and this method is called by the button clicked action listener.

e. *Robot Class*

As we did not think over updating the visible range deeply enough, we found that we cannot get visible range easily enough. Therefore, we add a `viewRangeList` field in Robot class to store the view range of a robot and to be updated when the robot is moved or died. And in the process of coding, we realized that we should reset the `movingPoint`, `hasShot`, and `hasPlayed` fields, we add a new method called `resetStatus` to reset those fields when a play starts. Also, we add the `relativeDirectionToCoordinate` method for the future UI display purpose.

iii. View Package

a. *GameBoardView Class*

We added `updateTimerNumber`, `updateMist`, `updateCurrentPlayer`,

updateCurrentRobot, updateOperationState, updatePlayerDeath, updateRobotLocation, updateRobotTurned, updateRobotDamaged, and updateRobotDestruction methods. In our previous design, those methods are included in the controller class. When we are in the construction phase, we realized that it is hard to implement updating UI if we put those methods in view. Thus, we move them from controller to view. Then the game data would be passed from controller to view by calling those methods in GameBoardView Class. Also, to make the view less coupled with the other parts of the project, we changed the type of data passed in view to basic types.

iv. Controller Package

a. *Controller Class*

We add five fields: playerNum, gameBoardViewTimer, operationMode, shootDistance and shootTarget. The field playerNum is to access the number of players when making choice of it on the second view. The gameBoardViewTimer is the system built-in timer, which controls the timing of each turn. In our design document we only designed a timer method in the Controller class, however, it is not enough. Thus, we add the timer field. The reason we add the other three fields is similar. We decide to store them separately to increase the readability of the code. Now other methods can use these fields' getter method as parameters instead of using a long method call. Also, we added some private helper functions such as initialGameWithMode, which make this class more structural.

For some methods such as updateRobotPosition(), updateRobotHealth() and updatePlayerDeath(), we move them to GameBoardView class. It will reduce code coupling between GameBoardView and Controller and make data transmission more efficient from Controller to GameBoardView.

v. aiutil package

a. *Interpreter class*

The class is to create an interpreter for the AI player. We discussed many times to develop this class, thus we almost rebuild this class with new methods to implement the interpreter class such as turn, move, shoot, etc.. Also, we write methods run and runStatement to receive the Forth script and translate it to Java. This reconstruction is much better than what we designed before that we can construct the AI player to run.

b. *AIEventListener class*

This class is a listener of AIEvent with a overridable method called EventActivated, which would be ran when the AIEvent is activated.

c. *AIEvent class*

This class is a custom event type for AIPlayer, which includes two fields, action and value. Those two fields could be got when handling the event as information needed. This is

vi. warofrobotcraft package

a. *Initial class*

We added the background music in this class. When the game begins, this background

music plays simultaneously until the game ends. This music brings more pleasure to the game and makes it less boring when other players are running the play.

vii. **Resources package**

This package is newly added to store resources including icons of the game board and the music used for the game. When we were constructing the game, we found that there were too many files needed to be used as materials, and seemed messy in the project folder. More importantly, their paths are not convenient to be used if scattered within the folder. Thus, we decided to create a new package to store them, and used the unified path that is "Resource/filename.suffix".

3. Bugs remain

As many tests we did, we have not found bugs within the runtime and compile time. Maybe bugs could appear in the future, the game runs smoothly till we hand in the project.