Design Document

Brady Lang

Dylan Mcinnes

Parker Reese

Meagan Renneberg

Table of Contents

Architecture …….. 2

Entity and Method Descriptions ……...…... 2

- Model ……. 2
- View ……5
- Controller ……5

UML Diagram ……...7

Changes Made from Requirement Document …….. 9

Architecture

We have chosen the architecture of model-view-controller for our project and the view components in our model-view controller is event driven. Due to time constraints we decided to use an architecture everyone in our group was comfortable with. This is an efficient architecture for our project because it best represents how we want our game to work. We wanted to be able to have the entity displaying the game be almost completely separate from the entity containing all the controls and data for the game. This idea was best solved with using a view for the display and a model for the containers, the main way the view and model ever interact with each other is through the controller. The controller in our system interprets the requests and information it is given and passes it on to the appropriate component based on what is being asked. The model in our system contains all the data and information about the game, so it is more efficient because we will only have to look at model components when looking for information rather than looking for it through the entire system. The controller in our system interprets the information given to it and then passes information and/or commands. This will make it easier to interpret actions done in the system because they are all interpreted by command components rather than the system constantly checking whether or not it can do every possible action. The view in our system controls what is displayed based on the information and commands it receives from the controller. This is a better way to display the game because there is one component in charge of displaying it rather than having multiple scenarios that would display the same thing. Our view is event driven internally because it uses the swing library which is almost exclusively event driven.

Entity and Method Descriptions

The components of the architecture as well as their respective methods are as follows:

Model

Gameboard:

The GameBoard is a model entity in our system. It keeps track of all the information pertaining to the game board. The GameBoard varies in size depending on the amount of teams in play. If two or three teams are playing, the game board will be size 5. If six teams are playing, the game board will be size 7. The methods **getSize()** and **setSize()** will

determine the correct board size and set up the game based on the correct board. The GameBoard also retains any statistics created during the game in progress. Examples of this would be a robot damaging another robot, or a robot getting damaged. These damages (both given and taken) need to be recorded and added to the robot library's statistics after the game. The method **updateImmediateStatistics(**RobotID, Stat, Value**)** handles the ongoing statistics in the game by taking in the Robot Identifier, which is unique to each robot, what statistic it should change, as well as how much it should change it by. The related method **getImmediateStatistics()** can be used to grab any ongoing statistic in the game. The method **getAllRobots()** returns a list of all robots playing in the game. Lastly, the method **isValid(**BoardPosition**)** is a boolean that checks if a specific board position is valid.

Robot:

The Robot is a model entity in our system. It holds of all the information pertaining to the robots of the system. Each robot (Sniper, Tank, Scout) have a unique *health*, *range*, and *damage*, all of which are expressed as member variables. Each robot will either be controlled by a human player, or an AI. An AI refers to the Script created to dictate the robots moving and shooting behaviours. To see what is controlling a robot's playing, run **isAI()**. To acquire that robots specific AI script, run **getAI()**. Each Robot will be unique and thus has a way to uniquely identify itself: the *identifier* member. Lastly, when a Team of Robots are not being controlled by human players, but rather AI, they have the ability to communicate with each other. This communication will be achieved through accessing the *mailbox* member, which is a list of the messages the robot has received. The *variables* member is a map indexed by the string name of the variable, keyed to the value. This is used by the NpcController's ForthInterpreter to store the robot's user (AI) defined variables and their values.

BoardPosition:

BoardPosition is a model entity in our system. BoardPosition is responsible for knowing the position represented on the game board, as well as how it's current position correlates to another position on the gameboard. The **turn**(*Direction*) method changes the direction of the board position given a specific point of direction, number 1-5. The method **advance**() then moves the position it is currently in by one in the direction it is currently facing. Next, the method **directionTo(**BoardPosition other**)** will be used on another board

position to determine the direction required to change the current position to the *other* position, usually used in conjunction with **distanceTo**. The related method **distanceTo(**BoardPosition other**)** is the distance between the current position and the position *other*. Another method **immediateDirection(**BoardPosition other**)** returns the immediate neighborhood direction towards the position *other* (if ambiguous, decide pseudorandomly). Finally, the **getRelative(**int direction, int distance**)** method returns a new BoardPosition, which has the position relative to the current position dictated by the direction and distance argument.

Team:

The Team entity is a model within our system. A Team consists of three Robots (a Sniper, Tank and Scout) expressed as a list of these robots in the member variable robot. Each Team will be unique and thus has a way to uniquely identify itself: the *identifier* member. When a game is created, each team that is part of the game is assigned a random unique color to be played, achieved by **setColor(**color**)**. Color dictates both starting position on the board and their respective turn orders Also, the colors available will be dictated by the number of players in the game, so that all teams start in the correct starting positions. **getColor()** returns the team color, used in board positioning and turn sequences.

Robot Library Socket:

The RobotLibrarySocket is a model entity in our system. The RobotLibrarySocket is to act as the half-way-house between the rest of our system and the robot library. To update the statistics in the robot library, it will run the **updateStatistics(**Gameboard**)** method. This will take place at the end of every game played. The GameBoard contains the statistics from the game played and thus is passed into the method to be used. To request a list of all of the Teams or Robots contained in the robot library the **enumerateTeams()** as well as the **enumerateRobots()** methods can be used, respectively. It may be of necessity that only one specific Robot or Team be accessed from the robot library, in which case the methods **retrieveRobotFromLibrary(**RobotID**)** and **retrieveTeamFromLibrary(**TeamID**)** can be used. It should be noted that the Robot Library only sends and receives JSON files, so it is the Robot Library Socket's job to create and translate these files in its requests.

View

GameDisplayer:

The GameDisplayer is a view entity in our system. This entity takes care of all the visuals during the playing of a game. Using the method **displayGame()** it will show the gameboard, and any teams, or robots necessary in their appropriate positions as well as directions. With the **close()** method it will exit the screen of the game.

SetupDisplay:

The SetupDisplay is a view entity in our system. This entity takes care of all the visuals for the GUI's during the setup of the game. This entity will display a GUI with the **displaySetup()** method, which brings forth a particular GUI dependant on the specific button clicked by the user. The **close()** method then exits the GUI needed.

Controller

Referee:

The Referee is a controller entity in the system. This entity manages the game while it is being played. The **startGame(**GameBoard**)** method will begin the playing of the game, allowing the red team to begin their first round. The (hopefully infrequently used) method **abandonGame()** will stop the game in progress. This will be useful for testing as well as error handling. The **updateGameBoard()** method calls the GameDisplayer entity to update the game display. This will take place after a change in state of the game has occurred, such as a movement, or a death of a Robot. Since the RobotController needs to acquire the GameBoard, the Referee has the method **getGameBoard()**, which returns the GameBoard. Finally, the Referee also handles the case of when a Team decides to leave the game. The **teamQuit(**Team**)** method will remove said Team from the game. The game, given that there is still more than one Team on the board, will continue to play as normal, minus the quitting Team's robots.

Game Initializer:

The Game Initializer is a controller entity in the system. **initializeGame()** will return a Gameboard. All the information given by the user during the SetupDisplay will be passed to

the GameInitializer, it will run **initializeGame()** with this information and create a Gameboard to be used for the playing of the game.

Note: The AI scripts for the NPC Robots, if any, are only loaded into the Forth interpreter after all the Teams and their corresponding Robots have been selected for the game and the game is about about to begin.

RobotController:

The RobotController is a controller entity in the system. It manages the actions of the robots during the game. Since the robots have the capabilities to play, quit, scan, shoot, and send a message to other robots, the RobotController then has to manage these. **play()** is called when it is a specific robots' turn to do actions, and all of it's actions are done inside play. The **quit()** method sends a message to the RobotController to remove the team that that robot is on from the game board. **scan()** is used to see what robots, both enemy and team, are within visual range of the playing robot. The robots that are in the range are then visible to that robot and displayed on the board. **shoot(**int distance, int direction**)** is used to shoot a space. This is done by passing in the distance from the robot to the space wanted to shoot on, as well as the direction between the robot and the space wanted to shoot on. NPC robots have the ability to talk to one another, when a robot wishes to send a message it uses **send(**Robot, String**)**. This method, given the specific robot it wants to send a message to, and the string containing a message, will add the string to that robot's mailbox. RobotController has a *myRobot* variable, containing a Robot so it knows which robot to take these actions on. It also has a *myReferee*, containing the Referee, in order to call it's methods when needed, Such as **teamQuit(**Team**)** when a robot initiates **quit()**.

Script:

The Script is a model entity in our system. This entity contains the AI logic for the NPC robots. This logic is held as a Map of words, ands get run in **play()**.

ForthInterpreter:

The ForthInterpreter is a controller in our system. This entity, given a Script, and after translating the Script, executes the given code.

PcController

A PcController, as mentioned in the UML, describes a human controlled player that interacts with the system.

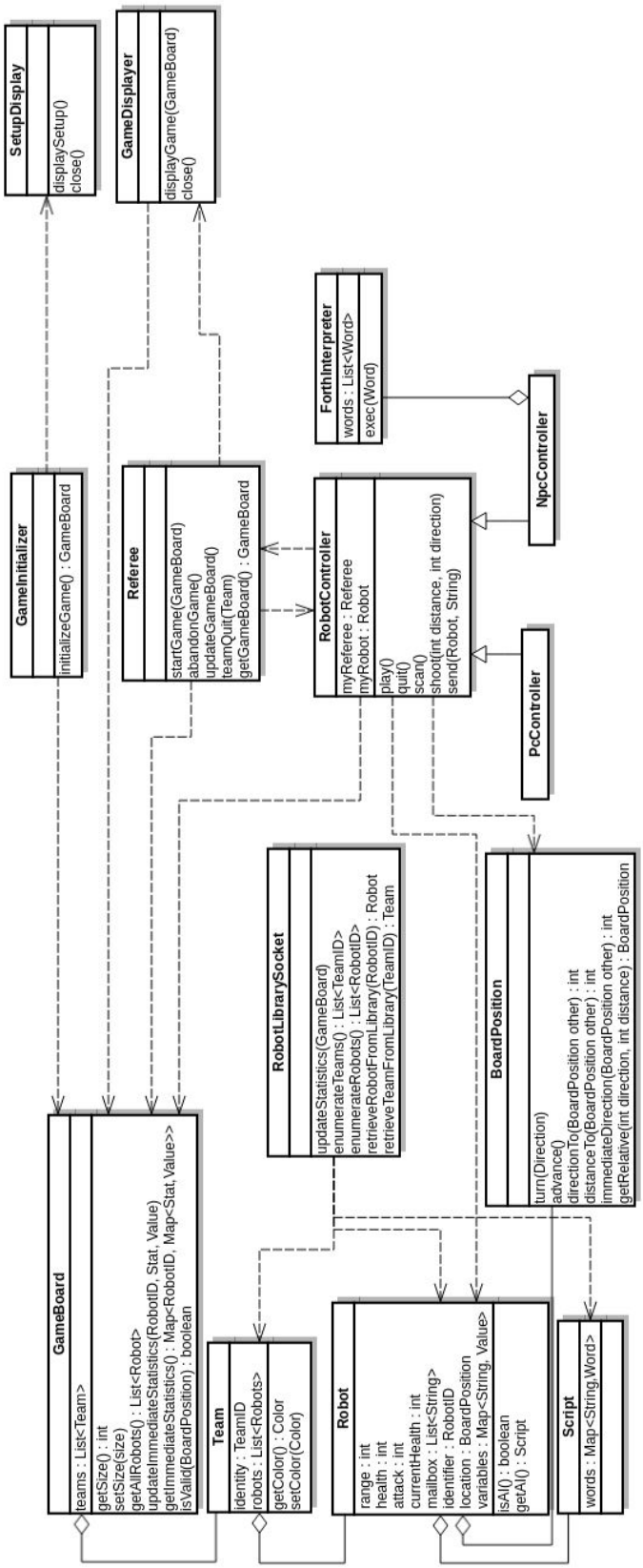
NpcController

A NpcController, as mentioned in the UML, describes a non-human interaction with the system.

UML Diagram

The UML diagram that displays the components of our architecture, as well as their interactions with each other, follows. See page 8.

UML Diagram


SetupDisplay
displaySetup()
close()
GameDisplayer
displayGame(GameBoard)
close()
ForthInterpreter
words : List<Word>
exec(Word)
NpcController
GameInitializer
initializeGame() : GameBoard
Referee
startGame(GameBoard)
abandonGame()
updateGameBoard()
teamQuit(Team)
getGameBoard() : GameBoard
RobotController
myReferee : Referee
myRobot : Robot
play()
quit()
scan()
shoot(int distance, int direction)
send(Robot, String)
PcController
RobotLibrarySocket
updateStatistics(GameBoard)
enumerateTeams() : List<TeamID>
enumerateRobots() : List<RobotID>
retrieveRobotFromLibrary(RobotID) : Robot
retrieveTeamFromLibrary(TeamID) : Team
BoardPosition
turn(Direction)
advance()
directionTo(BoardPosition other) : int
distanceTo(BoardPosition other) : int
immediateDirection(BoardPosition other) : int
getRelative(int direction, int distance) : BoardPosition
GameBoard
teams : List<Team>
getSize() : int
setSize(size)
getAllRobots() : List<Robot>
updateImmediateStatistics(RobotID, Stat, Value)
getImmediateStatistics() : Map<RobotID, Map<Stat, Value>>
isValid(BoardPosition) : boolean
Team
identity : TeamID
robots : List<Robots>
getColor() : Color
setColor(Color)
Robot
range : int
health : int
attack : int
currentHealth : int
mailbox : List<String>
identifier : RobotID
location : BoardPosition
variables : Map<String, Value>
isAI() : boolean
getAI() : Script
Script
words : Map<String,Word>

Changes Made from Requirement Document

The changes we have made from our requirements document are that the robot librarian is no longer a main actor in our system. Rather, it is a separate system that is interacted with by a new entity in our system, namely the Robot Library Socket. Also, we are no longer allowing users to edit, create and change their robots. All Robots will be pre-made into the Robot Library. We have decided to remove this interaction with the Robot Librarian due to it not being a necessity for our game to function. User ability to initiate interactions with the Robot Library are now simply a “nice to have”.