**ARCHITECTURE AND**

**DESIGN**

**Robot Arena**

**GROUP D1, CMPT370**

**Nico Dimaano, ned948**

**Niklaas Neijmeijer, nkn565**

**Kyle Seidenthal, kts135**

**Brendon Sterma, bws948**

**Jiawei Zang, jiz457**

**INTRODUCTION**

**Purpose**

The purpose of this document is to outline the architecture and detailed design of the Robot Arena system. The Robot Arena system is a multiplayer turn based game that will feature online multiplayer and non-human players if desired.

**Scope**

This document describes the information needed to implement the Robot Arena system. It will contain a description of the architecture as well as a detailed level class design.

**SYSTEM ARCHITECTURE**

**Chosen Architecture**

The architecture we have chosen for this project is the “Model-View-Controller” architecture. This architecture works well with the concept of a game, and allows us to separate the interface from the game model. “Model-View-Controller” keeps the user interface separate from the game logic, which makes it easy to design, update, and maintain the interface to the game. By keeping the game logic in one place, any changes to the game logic itself are also easier to manage as they will not affect the controller or view. This architecture will also allow for easier testing, as each layer can be tested separately and can be guaranteed to work with any other layer through predefined interfaces. Overall the “Model-View-Controller” matches the system requirements more effectively than other architectures we have studied.

**Other Considerations**

We have studied into a few different options for the architecture of our system to ensure we maximize the efficiency of the system. The following section outlines the details of some of our options. The major architectures we considered were: “Data-Driven”, “Call and Return”, and “Pipes and Filters”.

One of the architectures we studied was the “Data-Driven” architecture. This does not seem to be a necessary architecture for our system as there is no underlying data storage or lookup. All data used in the system is created and used during play and is not stored after the application has been terminated. The robot data from the librarian will travel through the system in JSON format, however it is unnecessary to store this information after a match has finished as it will be sent back to the librarian, which will manage it per its specifications. Thus, a data-driven architecture seems to have more overhead than is necessary for the system, and would ultimately be more costly than beneficial.

We also studied the “Call-and-Return” architecture. Some aspects of this architecture could be beneficial to the system, such as the ability to easily distribute across multiple machines or networks. Though this may make the networking pieces of the system easier, “Call-and-Return” has negative affects to keeping the game system simple and easy to manage. The hierarchical nature of this architecture would not allow us to abstract the input and output interfaces from the model of the game, and many objects would rely on each other, creating high coupling. This will ultimately impede our ability to manage game components separately from the user interface, making testing and implementation more work than is necessary. The “Call-and-Return” architecture does not fit well with the requirements for the system.

The “Pipes and Filters” architecture was the final architecture we considered. We found that this architecture would be useful for translating information from one state to another. It is not particularly useful for the overall game system, but may prove useful for specific pieces in the architecture. As an example, the robot librarian collects robot programs and gives them to the system. This data needs to be parsed out into a format that can be read by the system. Using “Pipes and Filters” would be useful in designing this piece of the system, but will not be effective for the overall game, where there is really no pipeline of commands being executed.

After exploring the above options, we feel we have adequately justified the use of the “Model-View-Controller” architecture for our system. It will allow us to separate the interfaces and maintain the system easily. All of the game logic will be kept in one place and will also allow for easy testing. As the system is meant to implement a game, the ability for the architecture to lower coupling for the user interface and game model makes “Model-View-Controller” an excellent choice.

**DESIGN OVERVIEW**

**Overview of the System**

*The above figure shows an overall diagram of the system. It is divided into three layers, the View is the first layer, the Controller is in the center, and the Model is on the bottom. Some subclasses and other classes have been removed for readability.*

The Model-View-Controller architecture allows the view to communicate with the model through the controller. This decreases coupling for the system. Each level of the system is described in detail below.

**Model**



*The above figure shows the class relationships in the model. The robot and tile classes are interacted with by the user, while the controller acceses the server, librarian, and AI.*

The AI Processor contains a copy of an AI File, a legend and a translate function. The legend consists of a list of Pairs. Each pair contains a Fourth function from the AI File and a java function that mimics the behavior in the java environment. The translate function finds a Pair in the legend that contains a desired java function. It then runs the co-responding action in the AI File, written in Fourth. The result is recorded and matched with another pair, executing the Java action associated with it.

Robot Librarian Processor contains a list of Robots and four functions for modifying them. The first is adding a robot to the list. This uses the list class's own adding functions as per choses implementation. The remove robot function also uses the list's functions to remove a specific robot from the list permanently. Clearing the list removes all robots from the list in an identical manner. The getList() function allows for the robot list to be returned as a full list to be manipulated by other exterior functions.

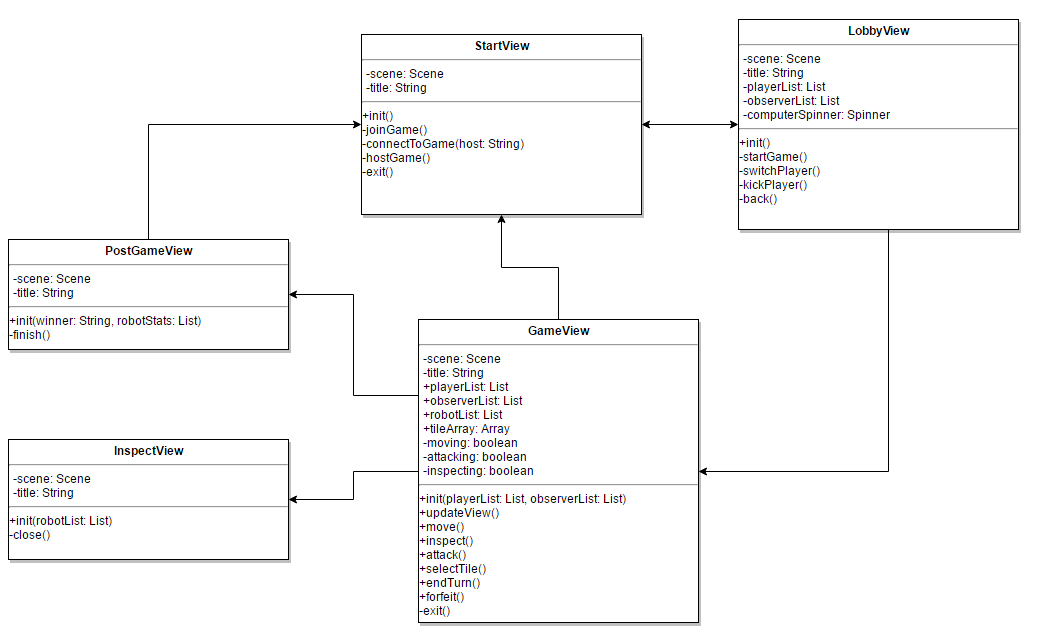
The Robot Librarian Processor contains a list of Robots and four functions for modifying them. The first is adding a robot to the list. This uses the list class's own adding functions as per choses implementation. The remove robot function also uses the list's functions to remove a specific robot from the list permanently. Clearing the list removes all robots from the list in an identical manner. The last function allows for the robot list to be returned as a full list to be manipulated by other exterior functions.

The tile class consists of a list of robots, and functions to manipulate them. A robot can be added to the list, a specific robot can be removed from the list, and a specific robot can be returned from the list for use by an external function. The entire list can also be returned in the same way. A specific robot can be updated by replacing that robot with an updated version.

The Robot class represents a robot in the game. Robots have many properties that allow them to be distinguished from one another, such as name, team, and their movement and attacking stats. Robots come in three types, Scout, Sniper, and Tank. These subclasses determine the stats for the robot. Robots can move between tiles, and can attack other robots through the respectively named functions.

The User class represents a human player for the game. The user has a Name conditions to determine whether they are the host of a match and if they are currently in a match. They have getters and setters for all of these properties. They have two subclasses. The first is the player, who can use the inspectTile() function to inspect a tile, the moveRobot() function to move robots, and the attack function to attack an enemy robot. They may also end their turn with the endTurn() function and forfeit the match with the forfeit() function. The player class has a list of their robots. The second subclass of User is Observer, which can use inspectTile() to look at any tile on the game board. This differs from the user’s getTile() in that the user can only inspect tiles in their view. The observer can also leave the game.

**View**



*The above figure shows the View level of the system.*

The StartView class represents the screen shown to the user when the program begins. It has a title and a scene to display. The StartView is initialized via the init() method. It has three options represented by its methods: joinGame, which allows the user to specify the address of an active match, hostGame which allows the user to host a match, and exit which allows the user to exit the program.

The LobbyView displays the game lobby to the user once the user has joined the match. The view has a title, a list of players, and a list of observers. The startGame() method starts the game and starts the GameView. There is a switchPlayer method that allows a user to become an observer, or an observer to become a player. There is also the kickPlayer() method, which removes the desired player from the match.

GameView:

**Variables:**

* -scene: Scene
  + The scene for the starting view
* -title: String
  + The title displayed across the top of the screen
* +playerList: List
  + A list of the players still in the game
* +observerList: List
  + A list of the observers watching the game
* +robotList: List
  + A list of all the robots still alive in the game
* +tileArray: Array
  + An array containing every tile on the board
* -moving: boolean
  + A variable that determines if the player moves when they click on a tile
* -attacking: boolean
  + A variable that determines if the player attacks when they click on a tile
* -inspecting: boolean
  + A variable that determines if the player inspects tiles they click on

**Methods:**

* +init(playerList: List, observerList: List)
  + Method that creates all visual elements for the users to see. playerList and observerList are used to populate GameView’s Lists and determine map size.
* +updateView()
  + Method that updates the map after every action
* +move()
  + Method that sets the moving variable
* +inspect()
  + Method that sets the inspecting variable
* +attack()
  + Method that sets the attacking variable
* +selectTile()
  + Method that informs the controller what tile was clicked, and which of the three actions (moving, attack or inspecting), was currently true.
* +endTurn()
  + Method that informs the controller that the “End Turn” button was clicked
* +forfeit()
  + Method that informs the controller that the “Forfeit” button was clicked.
* -exit()
  + Method that informs the controller that the player has closed the game

InspectView

**Variable:**

* -scene: Scene
  + The scene for the starting view
* -title: String
  + The title displayed across the top of the screen

**Methods:**

* +init(robotList: List)
  + Method that creates all visual elements for the users to see. robotList is a list of the robots on the tile being inspected
* -close()
  + Method that closes the InspectView window

PostGameView:

**Variables:**

* -scene: Scene
  + The scene for the starting view
* -title: String
  + The title displayed across the top of the screen

**Methods:**

* +init(winner: String, robotStats: List)
  + Method that create all visual elements for users to see. winner is displayed as a label, while robotStats is used to populate a table.
* -finish()
  + Method that returns you to the starting screen.

**Controller**

**Server interface:**

which has few functions which are connects a server that needs to keep track of the state of a game, and inform each client of it, so they can each update their own displays. and this interface will contain

getname

Prompt for and return the desired screen name

Run

this function will connects to the server

GetserverAddress

This is function which will Prompt for and return the address of the server

**Model Interface:**

which will has function which the model element will using

getname

this function will get the string the objects

**AI Interface:**

is the function will calculate from the "battle field" or combat information in the game

and this function

**ROBOT Interface,**

setname

set the name

setteamname

set the team name

setmatch

set the match

setdied

set the died number

setlived

set the live

sethealth

set the health

setfirepower

set the firepower

setkill

set kill number or count

setmovement

set the movement

setdistance

set the distance

setfirstRun // check condition

check which one is first

getname

return name

getteamname

return team name

getmatch

return match

getdied

return died

getlived

return lived

gethealth

return health

getfirepower

return firepower

getkill

return kill

getmovement

return movement

getdistance

return the distance

getfirstRun // check condition

return who is the first Run

**Game:**

this is class will mainly initialize the game, and handle the user input

which will contain functions such as:

init

which will initializes the views

RGHexboard

initializes the board when we press start button

changeScreen(MouseEvent)

change the screen when we press button

initview

draw game when we enter the game

getview

show minimap

inputdetecttor(button)

changing the windows, which we getting hit

checkwinlose

checking win or lose, if lose or win it will shows the windows which is winloss windows view

Main(String [] )

launchApplication

**REQUIREMENTS TRACEABILITY**

We have decided to make minimal changes to our requirements. We have made the robot librarian an external system to our own, and added a win lose screen to our view.

**Version History:**

* **23/10/2016 – Initial DESIGN version – V1.1**