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.

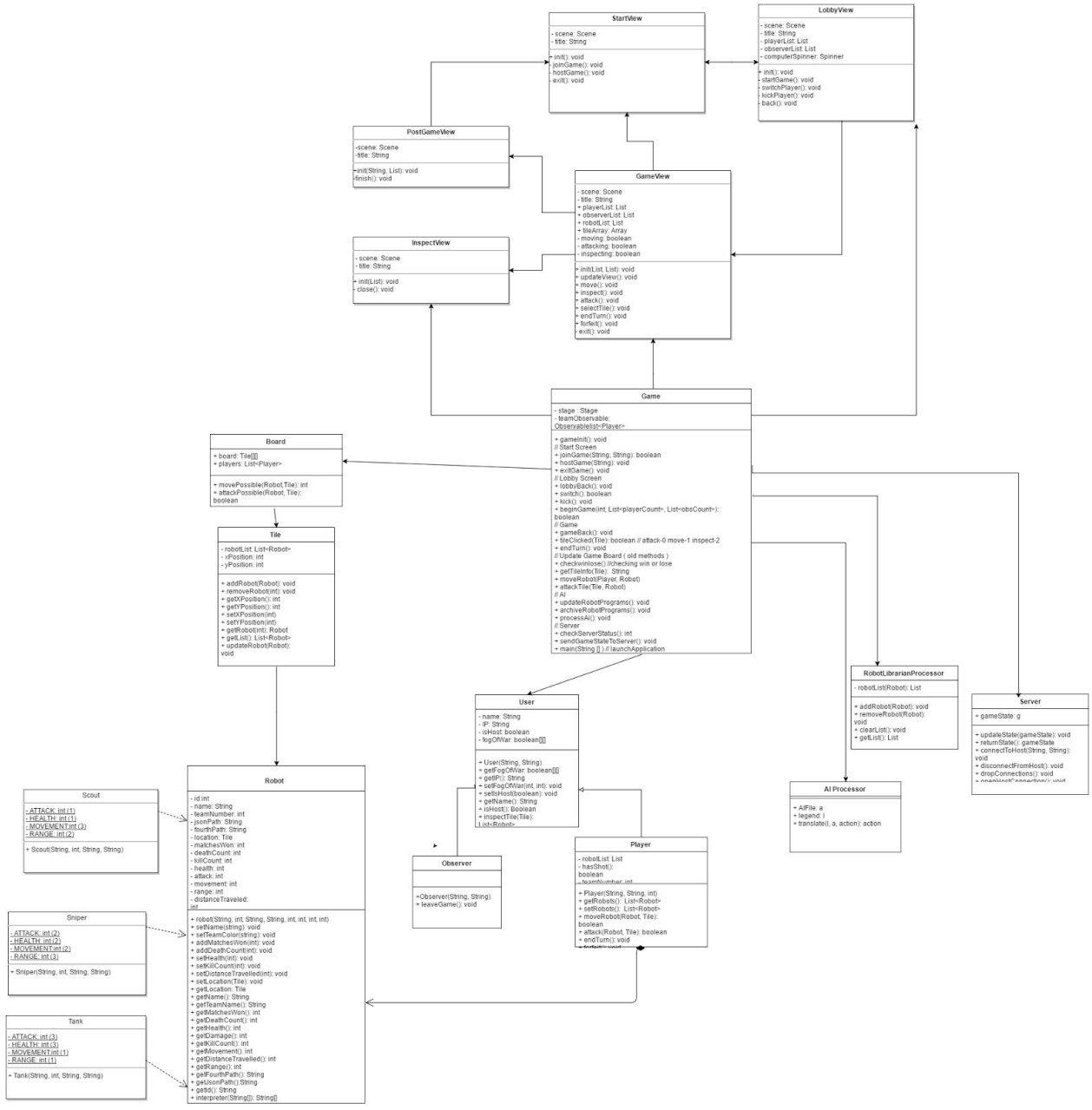
We also studied the “Call-and-Return” architecture. Some aspects of this architecture could be beneficial to the system, such as the ability to easily distribute across multiple machines or networks. Though this may make the networking pieces of the system easier, “Call-and-Return” has negative affects 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.

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 class is interacted with by the user, while the controller accesses the board, 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 corresponding 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.

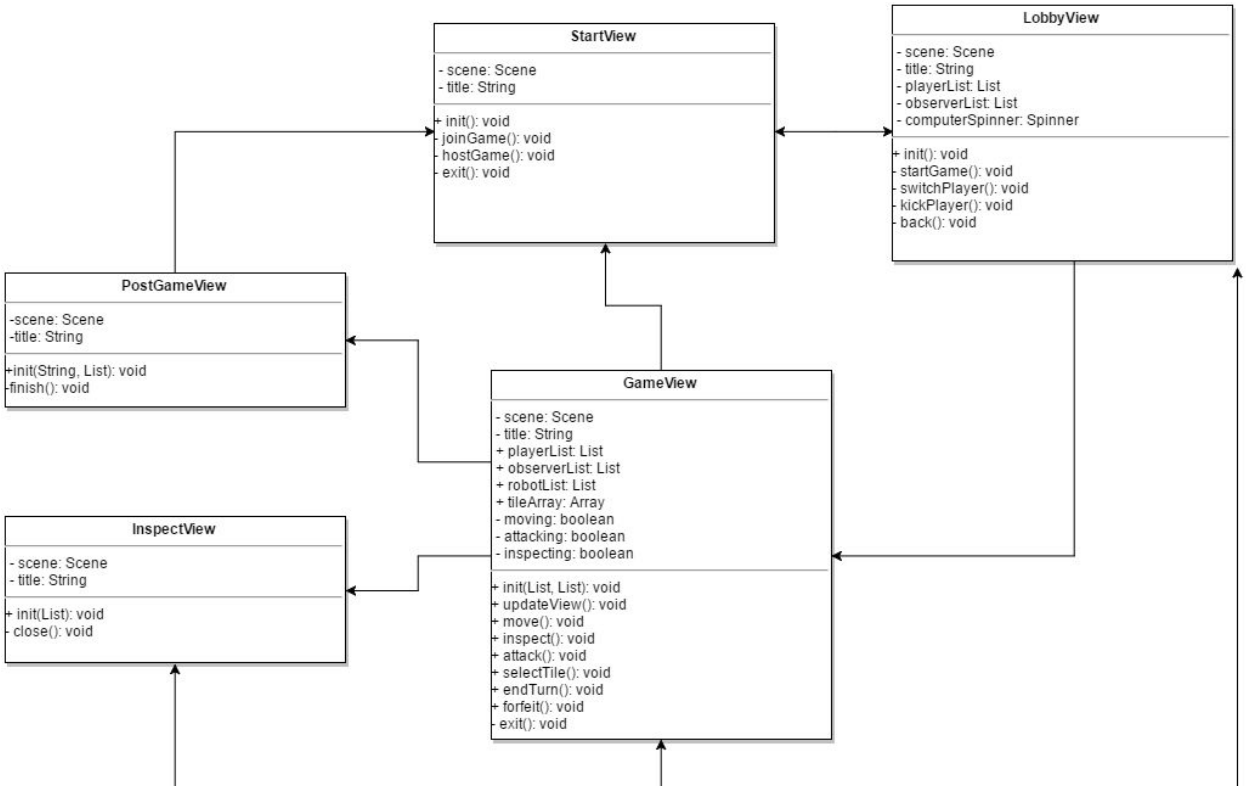
The Board class consist of a two-dimensional array of Tiles, a list of Players and methods to compare tile locations. The first method, movePossible is given a Robot and a Tile, and will get the the tile the robot is on and calculate if the robot has enough movement to go to the destination, returning the number of moves it will take if it is possible, and -1 if it is not. The second method, attackPossible is also passed a Robot and a Tile, and compares the distance between the Robot's tile and the passed in Tile. If it is less than the Robot's range it returns true, otherwise it will return false.

The Tile class consists of a list of robots, and its own x/y coordinates on the Board. It possesses getters and setters for both of its' coordinates, and two functions that return Robots. The first, getList, will return the robotList that Tile stores, the second getRobot, will return a Robot that has an id that matches the passed in int, if it has one. It also possesses methods to add, remove or update the list of Robots that it stores.

The Robot class represents a robot in the game. Robots have many properties that allow them to be distinguished from one another, such as id, name, team, Tile, and their movement and attacking stats. They also store the absolute path to Robots come in three types, Scout, Sniper, and Tank. These subclasses determine the stats for the robot. The only methods that Robot has are getters for all of it's attributes, setters for the ones that may change and a method to interpret the fourth commands.

The User class represents a human or AI player or observer for the game. The user has a Name attribute which is displayed during the game, an IP attribute to know how to reach that user, a boolean attribute to determine if they are the host of the game and a two-dimensional array of boolean values to determine what parts of the board that user is able to see. The user possesses The ability to get and set all of it's values except for the IP address, which is defined when the user is constructed. It also has the method inspectTile, which takes in a tile, and if it is within the user's line of sight then it returns a list of Robots on that tile. There are two subclasses for user. The first is Observer, which only has the leaveGame method to allow it to return to the starting screen. The second is Player, which stores a list of all Robots associated with that Player, a boolean value to determine if the Player has already shot or and a teamNumber, which helps determine player order once the game starts. It has methods to get and set the robotList, and the methods moveRobot and attack, which both take in the Robot that is moving or attacking and the Tile that is the destination. The last two methods are endTurn which starts the next Players turn, and forfeit which removes all the Players Robots and turn them into an Observer.

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.

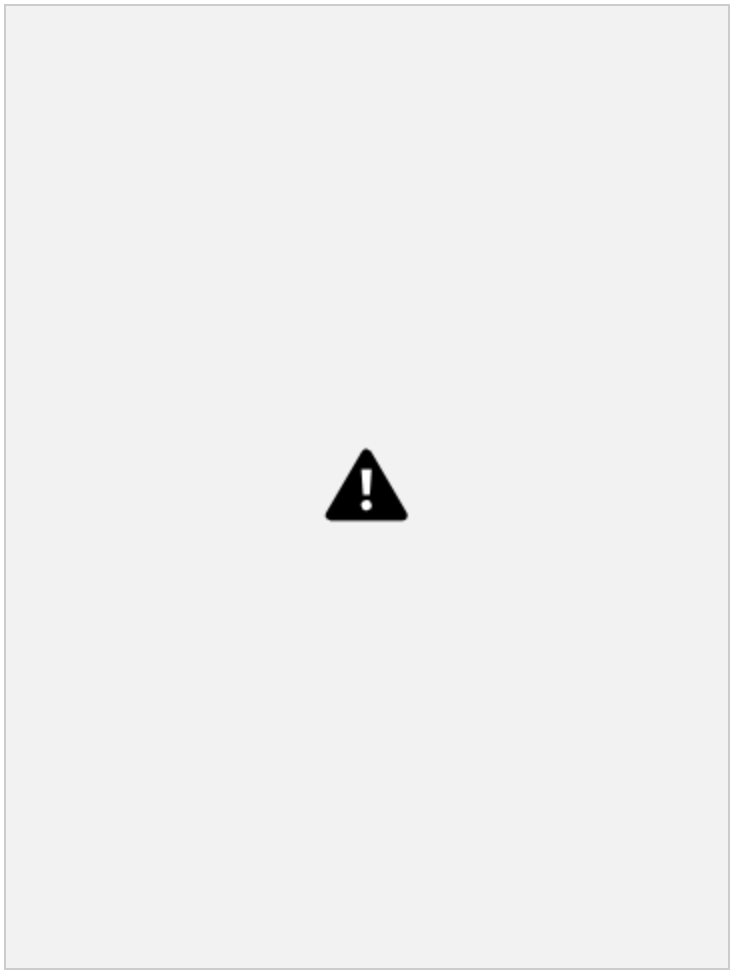
The LobbyView displays the game lobby to the user once the user has joined the match. The view has a title, a list of players, a list of observers and a Spinner tracking the number of AI Players. The startGame() method starts the game and starts the GameView if a valid number of players are present. 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 allows the host to remove the desired player from the match.

The gameView displays the current ongoing game. It has a list of players, observers, tiles, and robots in the game. It also has three boolean attributes, called moving, attacking and inspecting which are associated with the move, attack and inspect methods. Depending on which of the three aforementioned methods was last called, will determine what the selectTile method does when it is called. The game view has methods to communicate with the controller to move robots, attack other robots, inspect tiles, end turns, forfeit games and update the views of all current Players.

The InspectView is a popup called by the GameView. It has a title and a scene to start the view. It is initialized with a list of robots to display to the player. It is closed with the close function.

The PostGameView displays a screen to the player that shows the results of the match. It has a scene to start the view and a title. It is initialized with the winning player and a list of robots. The finish method finishes the match.

Controller



The above figure shows the Controller of the system.

The Controller

The Game class is the controller of our project, and contains the main method to launch the application. It has methods that are called when the view class's buttons are clicked, methods to check if a player has been eliminated, if the game has been won, communicate with the various model classes and to handle which Robot is currently active.

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:

- **4/11/2016 – Initial DESIGN version – V2.1**

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