**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 according to 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.

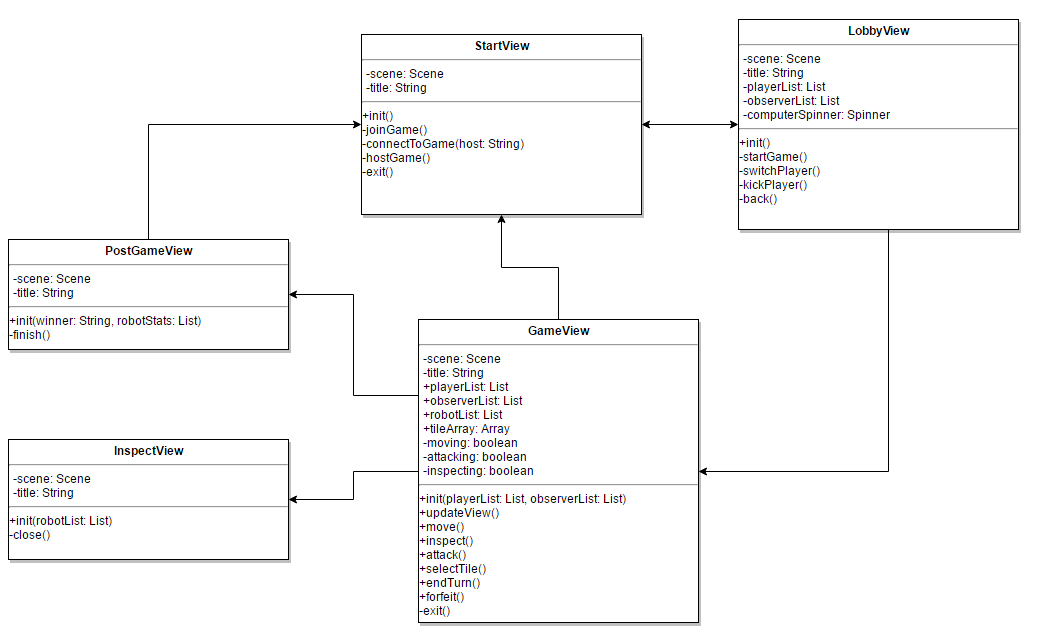
After exploring the above options, we feel we have adequately justified the use of the “Model-View-Controller” architecture for our system. It will allow us to separate the interfaces and maintain the system easily. All of the game logic will be kept in one place and will also allow for easy testing. 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**

**Model**

**View**



StartView:

**Variables:**

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

**Methods:**

* +init()
  + Method that creates all visual elements for the users to see
* -joinGame()
  + Method that opens an entry to type the address of the game you want to join. Calls connectToGame
* -connectToGame(host: String)
  + Method that takes in a String which is the address of the game to be connected to. Passes along to controller
* -hostGame()
  + Method that moves you to the lobby screen
* -exit()

Method that closes the program

LobbyView:

**Variables:**

* -scene: Scene
  + The scene for the lobby view
* -title: String
  + The title displayed across the top of the screen
* -playerList: List
  + A list of all users that are going to be players once the game starts
* -observerList: List
  + A list of all users that are going to be observers once the game starts
* -computerSpinner: Spinner
  + A spinner that tracks how many computer players there will be once hte game starts

**Methods:**

* +init()
  + Method that creates all visual elements for the users to see
* -startGame()
  + Method that tells the controller to start the game
* -switchPlayer()
  + Method that tells the controller to swap the user from player to observer and vice-versa
* -kickPlayer()
  + Method that is only available to the host, will remove the selected player from the lobby
* -back()
  + Method that indicates to the controller to return you to the starting screen

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**

**REQUIREMENTS TRACEABILITY**

**Version History:**

**-**