CMPT 370 Software Design Document for BattleBots

Mackenzie Power

Haotian Ma

Ryan Tetland

Will Revell

Mitchel Kovacs

Phase2_DESIGN

Table of Contents

Introduction

Purpose

The purpose of this design document is to present our architecture's description, plan the classes and their interactions that which will be implemented, and state any changes from our requirements document that must be made. The architecture section will state why we chose to implement our architecture, the advantages and disadvantages of its implementation, and why chose it over other architectures that we considered. In the Unified-Modeling-Language section, we will describe what each class should do, what information they will contain, and what information they will send to other classes. It will not contain any code specifications, just how the different classes are connected. Unified-Modeling-Language diagrams will be provided with detailed descriptions. After designing the architecture, we expect to come across newly discovered difficulties that we did not know when writing our requirements, and as a result will have to update our promises made in the requirements document. We will state the changes from our requirements and also state why these changes must be made. This document will describe the classes that will later be implemented, and show their interactions with each other.

System Overview

The architecture we will implement is Model-View-Controller. The view will receive messages from the controller of what it should be displaying to the user. It will also send messages to the controller when the user interacts with the view through buttons and key presses. It will not have any direct interactions with the model of the system. The model will take messages from the controller and perform the required task then send information back to the controller. These parts of this architecture along with their interactions will be described more in the architecture system section. For further understanding, specific definitions and acronyms used frequently in this document will be briefly described.

Definitions and Acronyms

Unified Modeling Language (UML): a diagram to show the function signatures, and entities of classes. It also shows the various relations between classes.
Model-View-Controller (MVC): design architecture that is divided into three separated parts that communicate with each other. The model manages the data and behind the scenes workings of the program, the view is the interface for the user, and the controller communicates between the other two.
JavaScript Object Notation (JSON): a data file that will be used to transmit robot scripts, and robot statistics into Java.
Robot Librarian: a system for access of a collection of JSON scripts that will contain statistics for robots and their script. It updates the robots statistics after the match is completed and allows for downloading and viewing of various robot teams and attributes.

Constraints

In our system, we will be using a given robot librarian system to access robot scripts and stats. This will require us to implement a coordinate system in a way that compiles the robot scripts. The coordinate system is relative to each robot. When a robot is facing a direction, straight ahead of it is always coordinate zero, and the hexagons one move away from it have a direction of zero though five in a clockwise direction. Spaces two moves away are labeled zero through eleven, and spaces three moves away are labeled zero through seventeen. The robot scripts will be encoded through JSON and require the robot librarian to display their stats. Communication between different robot players will be done through the programing language Forth and have to work with java in order to be executed. Robots will communicate through Words and an internal mailbox system.

Each robot has a given set of resources. This includes attack, health, movement, and range. These variables must be kept track of to determine when the robot is out of moves, when it is dead, how many spaces it can see, and how much damage is dealt out. The following is each robot with its resources: Scout has 1 attack, 1 health, 3 movement, and 2 range; Sniper has 2 attack, 2 health, 2 movement, and 3 range; Tank has 3 attack, 3 health, 1 movement, and 1 range.

Design Changes

Robot Librarian

After receiving feedback from our requirements document, we now realize that the robot librarian is not a part of our system because we are not implementing it ourselves. The robot librarian is an actor with its own system that interacts with our system, and will be given to us when we begin coding.

End turn for human

During the design process, we realized that we did not include an end turn option for a human player. This is important to have because a player may want to end their turn even if they still have moves or shots left. We will add a button in the game screen interface labeled "End Turn" that will end the player's turn and allow the next players' turn to start.

Out of Bounds

If a human or robot attempts to move to a location that is out of bounds then the robot will just stay in the hexagon that they are currently in.

Feature Decisions

In our requirements document we stated some functionality that the game "should have". Now during the design process we have decided to include a section on the side of the game play screen that displays each robots current statistics. We will also display the final statistics upon completion of each game. We also had a section in our requirement document with functionality that the game "could have". We have decided to not include any of these so we can focus on the more important functionality of the game. It is more important to have the game functioning properly than have a game with fancy attributes but does not work.

Architecture Design

Description:

The chosen architecture for designing and implementing the BattleBots program outlined in the requirements document will be a Model-View-Controller. The Model-View-Controller consists of three different parts: Model, View and Controller. The Model is responsible for accessing and maintaining the logic and data of the system. It is responsible for updating and displaying the state to View so the user can see what changes have been made as a result of the input they have given. The model does this indirectly through the controller so it never directly interacts with the view. The same is true for the view, it also does not directly interact with the model, but through the controller. Since the view is the output tool in this relationship, the controller will be the input tool where it interacts with the user by taking user's input from either keyboard or mouse and communicating the action to the Model.

Justification

There are a number of reasons why this architecture has been chosen over other types. The primary reason for this design choice is that all members of the A5 software team already have a familiarity with how this architecture should work. All members of the group have taken computer science courses which involved creating programs based on a Model-View-Controller architecture, so experience with this method is present. We acknowledge that generally a Model-View-Controller architecture is a more efficient architecture for projects that are much larger than this. However, we feel as though our experience with this architecture relative to other architectures outweighs any difficulties we may have in implementing such a small program with a Model-View-Controller. Another reason why this architecture was chosen is because of the heavy use of visual displays driven by the user (relative to other types of programs). This program is driven by the users' input. Everything from viewing the rules; selecting team attributes; choosing game properties; and executing individual moves require input from the user in order to be performed. Every action done by the user will require a change in display either as a new interface window or as a visual change in board state. There will be very few actions done without the user's input.

A benefit of this architecture is the separation of the interface and the model. This will make the code easier to work on because each class will have a specific purpose.

This will result in less coupling because changes made to a class should have minimal effect on other classes. Cohesive classes also result in much easier testing because each class can be tested individually.

Despite all these advantages, there are some disadvantages that we must acknowledge. One disadvantage with Model-View-Controller is it may be difficult to divide the work other than between the model, the view and the controller. This is because these parts of the architecture have many interactions within themselves. Another disadvantage is that it may take more time planning the interactions to work properly interact without having too much unnecessary coupling. We feel as though the benefits of a high level of cohesion achieved through this architecture will outweigh the time costs we may endure during designing and implementing the program. Other architectures were heavily considered during the initial planning stages of this document but where ultimately decided against. The reasons for this will be discussed in more detail.

Comparisons

MVC vs Pipe-and-Filter

Pipe-and-Filter contains individual programs transforming input data to output data. Based on what we have learned, we see pipelines as typically a unidirectional tool, which may cause us some unnecessary troubles for the later designing stages. Additionally, we know that Pipe-and-Filter has very high reuse potential but for us to look back in the individual parts will be a time consuming. MVC is relatively friendly for editing because most of the editing will be taking place within the model.

MVC vs Layered System

Layered System is effective for separating concerns by creating multiple layers, which reduces the impact of change when changes don't affect layer interfaces. However, because there may be too many layers, program performance will be degraded, especially when there are multiple players involved.

MVC vs Independent Component

To have all components work individually allows decoupling. This autonomy of components enhances reuse and evolution. This makes it much easier to introduce new components without affecting existing ones. However, the connections between components are not guaranteed. For example, components announcing events cannot be sure they are getting a response or have control over the order of responses. MVC on the other hand, will always allow the view to be updated through the controller based on what is in the model has.

UML Diagrams

The following section shows the UML diagrams for all the classes described in the program. They will visually show the various relationships between all of the classes in each of the model, view, and controller aspects of our chosen architecture. The last diagram will show how all three interact with each other. Descriptions of individual classes will be discussed in the next section.

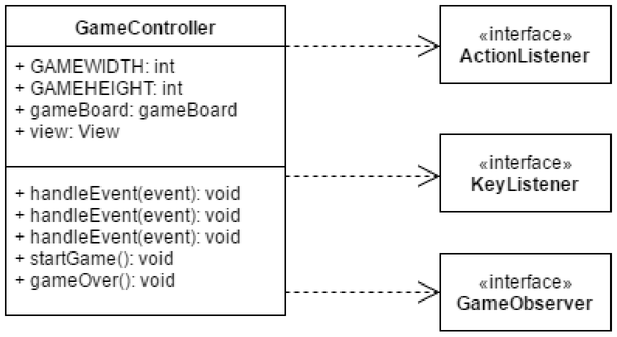


Figure 1. Controller UML Diagram. The Game Controller UML diagram is very simple as it simply contains the GameController class and the three different interfaces that it implements, ActionListener, KeyListener and GameObserver.

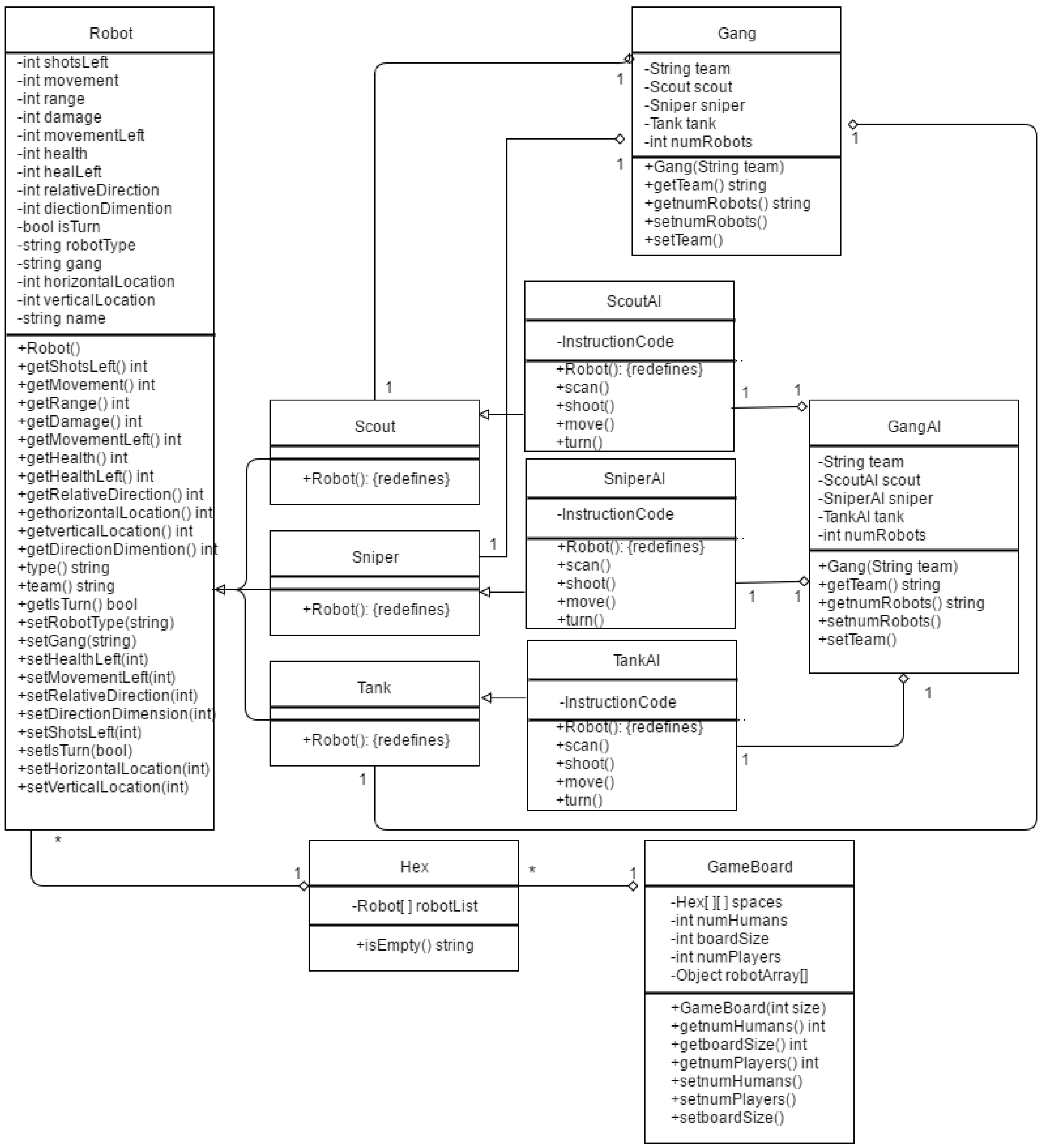
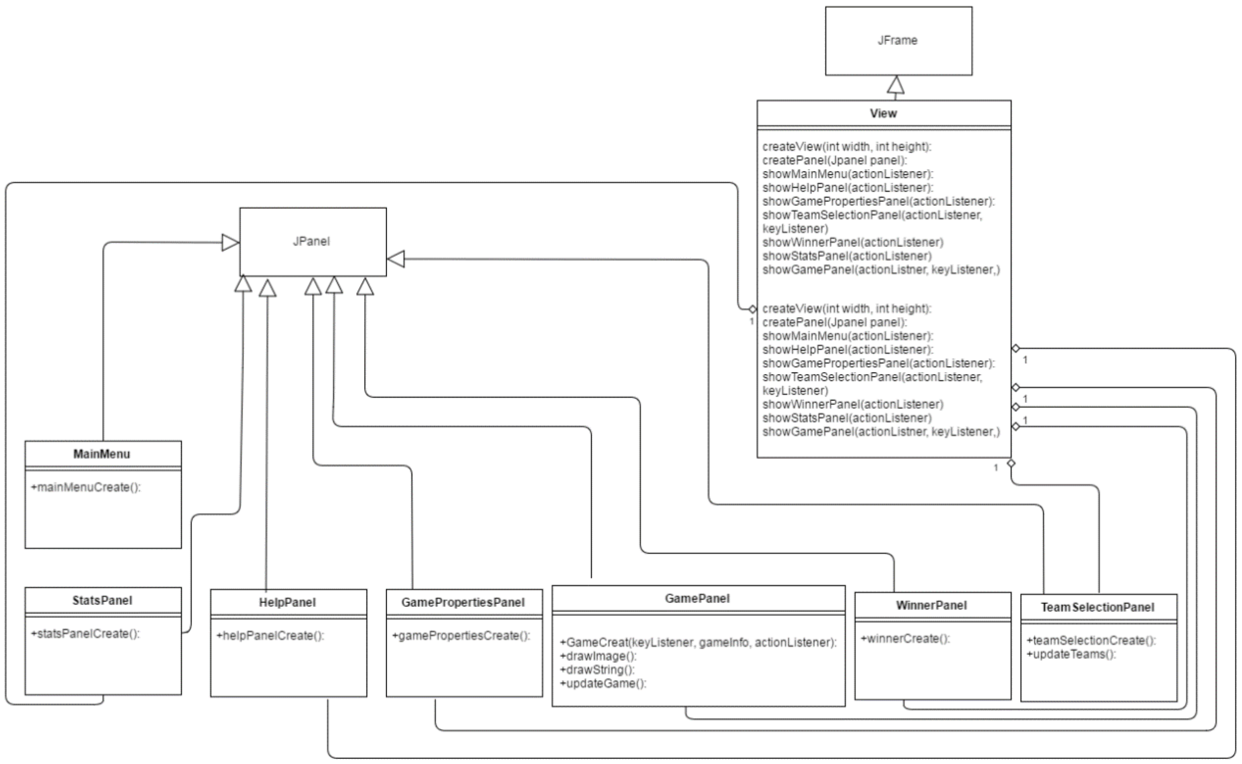


Figure 2. Unified-Modeling-Language Diagram for the Model of the Game's Primary Objects. This figure shows a few of the primary classes and their relationships that are part of the actual game design required to play. The Robot class has a list of several attributes and functions which will be required by all robot players regardless of whether they are played by robots or humans. The classes Scout, Sniper and Tank all extend the Robot class and their new overridden constructors will be tailored to instantiate them with their specific robot type attributes (such as health, range, etc.). Each one of these will further be extended to a ScoutAI, SniperAI and TankAI which are similar to their parents except that they will take in the JSON script provided by the robot librarian as an attribute and have the specified functions that robot players require to play. Each Scout, Tank, and Sniper will be associated with only one Gang class which will represent the

team that each robot is on. Each Gang will also only have one Scout, Sniper and Tank. Similarly, each ScoutAI, SniperAI, and TankAI will be part of one GangAI class which has one of each of those. The GameBoard class is literally the class that is representative of the game board players will play on. The GameBoard will be a 2 dimensional array of Hex's. For each GameBoard, there will be several Hex's which will be representative of spaces which can hold any number of Robots. Each Hex will be associated with only one GameBoard, and each Robot (or extended class) will be associated with one Hex.



JFrame
View
createView(int width, int height):
createPanel(Jpanel panel):
showMainMenu(actionListener):
showHelpPanel(actionListener):
showGamePropertiesPanel(actionListener):
showTeamSelectionPanel(actionListener, keyListener)
showWinnerPanel(actionListener)
showStatsPanel(actionListener)
showGamePanel(actionListner, keyListener,)
createView(int width, int height):
createPanel(Jpanel panel):
showMainMenu(actionListener):
showHelpPanel(actionListener):
showGamePropertiesPanel(actionListener):
showTeamSelectionPanel(actionListener, keyListener)
showWinnerPanel(actionListener)
showStatsPanel(actionListener)
showGamePanel(actionListner, keyListener,)
1
1
1
1
1
JPanel
MainMenu
+mainMenuCreate():
StatsPanel
+statsPanelCreate():
HelpPanel
+helpPanelCreate():
GamePropertiesPanel
+gamePropertiesCreate():
GamePanel
+GameCreat(keyListener, gameInfo, actionListener):
+drawImage():
+drawString():
+updateGame():
WinnerPanel
+winnerCreate():
Team SelectionPanel
+teamSelectionCreate():
+updateTeams():

Figure 3. View UML Diagram. Each of the classes MainMenu, HelpPanel, GamePropertiesPanel, GamePanel, WinnerPanel, TeamSelectionPanel, and StatsPanel extend the JPanel class. In addition, the GamePanel class will have functions to draw the current state of the game board. The team selection panel has functions to create the selected teams and to update teams. In the view class, it has functions to show each of the extended JPanel classes with actionListeners to display them at the appropriate time. The GamePanel and show TeamSelectionPanel 's have a keyListener so that they can receive input from the user.

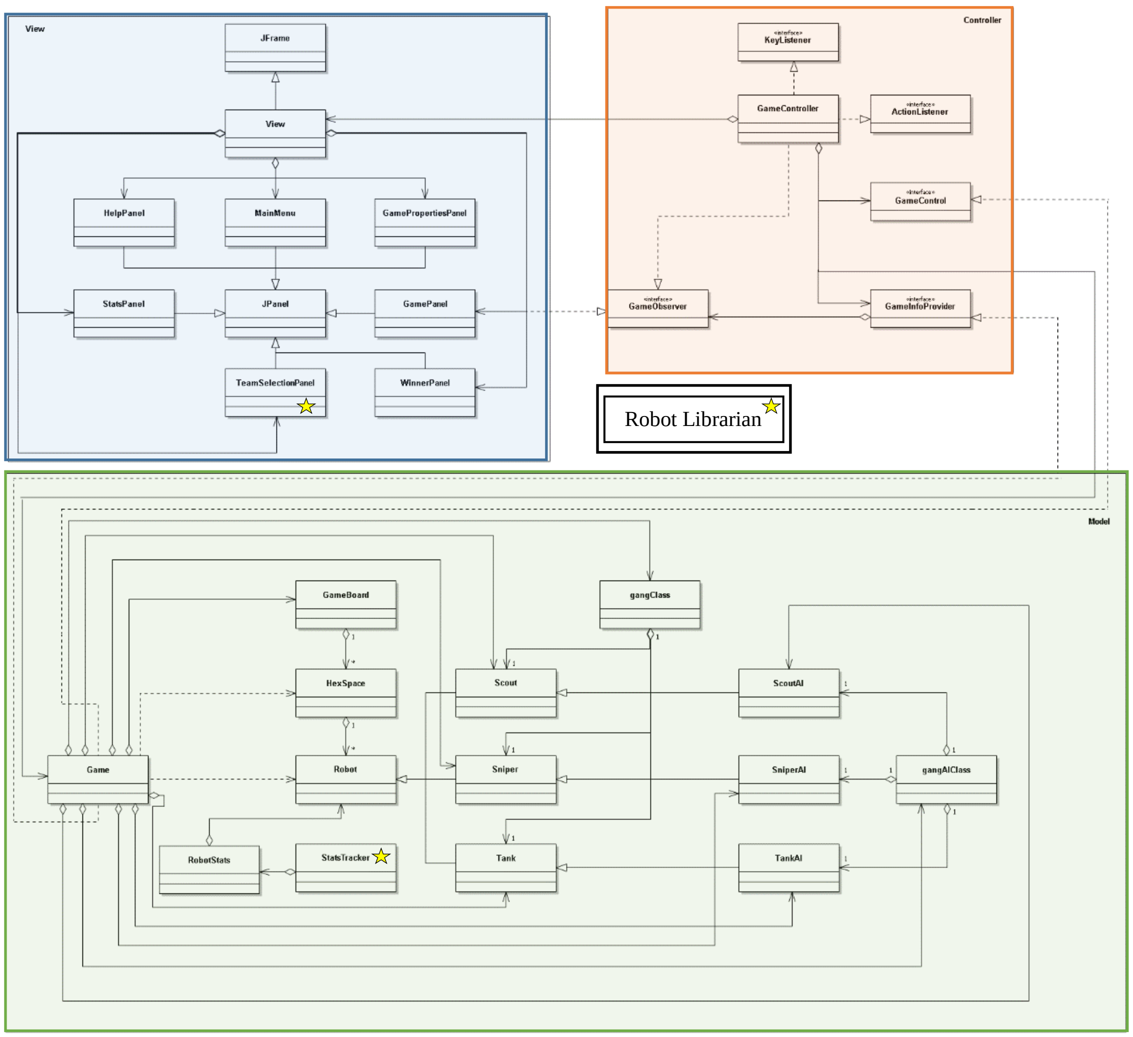


Figure x. Full MVC UML Diagram. The full uml diagram represents the culmination of the model, view and controller sections of the architecture and shows how they interact with each other. The view is perhaps the most separated of the 3 parts as it is accessed only through the GameObserver interface and by the GameController. The GameObseerver interface simply checks on the GamePanel whether the game is over or not, and the GameController updates the view with its updated information from the model which in turn updates the display. The Model also is only accessed by the Controller, specifically the Game class is accessed by the GameInfoProvider interface which passes it basic game creation information, the GameControl interface, and the GameController which deals with all user interactions and information updates to the Game class. The RobotLibrarian plays a small but critical role in the program. It is an external system that will interact with our system to provide robot script and statistics information. The primary areas of influenced are those desiganted with stars. The librarian must interact with the controller and TeamSelection panel to download and display robot information through JSON files. After a match has completed, it must again interact with a StatsTracker to update the JSON files of robots that were executed for that match.

CONTROLLER

GameController (controller)

This implements ActionListener, KeyListener, and GameObserver. The GameController class allows user and robot input on the games models and updates the display (view) accordingly. The GAMEWIDTH and GAMEHEIGHT attributes are global variables representing the desired size of the game screen. While the GameBoard and view will be used to create a display and create a populated game board . There are three separate handleEvent functions. The first is utilized only during the game screen and allows the user to move the robot using the AWEDXZ keys on the keyboard. The second is also only utilized during the game screen and allows the human player to shoot other robots by clicking on them with the mouse. The third and final handleEvent function is used to navigate throughout all the game screens and the game itself using the mouse to click on the navigation buttons such as continue and quit. The startGame function creates a game screen of a set size and takes the user to the MainMenu panel, the games beginning. The gameOver function will be consistently called during the game and will check if the game over state has been achieved and if it has it will end the game and take the player to the corresponding screen.

GameObserver<interface>

The GameObserver is an interface created to track the game state and see if the game has ended.

MODEL

GameBoard (model) implements GameInfoProvider. The GameBoard class represents the hex grid on which the game is played, it will utilize a 2-dimensional coordinate system with the center of the board being the origin (0, 0). The Hex attribute is an array that contains every hex on the game board with its corresponding coordinate value. The numPlayers, numHumans and boardSize attributes contain the basic integer values needed to create and populate the gameboard, while the robotArray contains each robot and its corresponding team. There are two main functions, the getRobots function creates a list of every robot that started play, and the GameBoard function creates the gameboard with the correct board size picked in game properties, as well as get and set functions to allow manipulation of the boardSize, numHumans, and numPlayer attributes.

GameInfoProvider<interface>

The GameInfoProvider interface allows for a cleaner and easier access to the games basic properties needed to run it.

Hex (model)

The Hex class will be the individual hexagon spaces that make up the game board. Each space must be able to store a robot of any type (scout, sniper, and tank) in an array. The amount of Hexes will be initialized by the GameBoard Class. It includes the attributes isEmpty which will be true if no robots are on the hex, and Robot array which is an array that contains every robot on the hex.

Robot (model)
The robot class will represent the basic robot type and will contain attributes used for every robot subtype, including; the robots movement range, attack range, shot damage, remaining movement left, the direction relative to the game board, if it's the robots turn or not, the current directions dimension it is facing (whether it is 1,2 or 3 squares away depending on the range) its type(scout, sniper or tank), the horizontal location coordinate, the vertical location coordinate, its team or gang and its current health value. It also contains several important functions including; a basic create Robot function that creates a robot object with attributes set to default values, it also contains a getvalue function for each of its attributes allowing access to each attributes value and a setvalue function for its health, direction, direction dimension, isTurn, robot type, gang/team, horizontal direction, vertical direction, shots left and movement allowing the values to be incremented or changed.

Scout (model), Sniper (model), Tank (model)
These classes extend the Robot class. Each of the Scout, Sniper and Tank classes extend the basic robot class and will be the robot objects controlled by the games human players. They all have the four attributes movement, range, damage and health from the basic robot class, but each is now set to the robot types default values. An updated create function now makes a new specific type robot with the appropriate attribute values.
SniperAI (model), TankAI (model), ScoutAI (model) extends Scout class
The ScoutAI, SniperAI and TankAI classes extend the Scout, Sniper and Tank classes and add five new functions. Move which takes in a target location and changes the robots' current location to those coordinates. The create function which takes in a string of AI instructions and uploads it to the new robot object. Scan which takes in the robots' current location and checks all hexes within the robots range for enemy and allied robots. Shoot which takes in a target location and deals the robots attack damage to all robots on that hex. And Turn which takes in the desired direction as a parameter and turns the robot relative to the game board.

Gang (model)
The Gang will represent each team color on the game board. It contains the attribute team representing its team color and an attribute for the Scout, Sniper and Tank robots which will be on the same team. The final attribute is numRobots which shows the number of remaining robots on the team. The Gang function takes in the team's color as a parameter and creates the corresponding scout, sniper and tank. There are basic get functions for the team and numRobots function which allows for easy access and a set function for numRobots allowing editing to show when robots are killed.

GangAIClass (model)
The GangAI is identical to the Gang save for the ScoutAI, SniperAI and TankAI attributes which have been changed from Scout, Sniper and Tank. The create function will now create the AI classes instead of the human controlled robots.

RobotStats (Model)
The RobotStats class tracks the stats of each robot during the game and keeps track of the attributes kills, deaths, games played, games ended alive, wins, loses, damage inflicted, damage absorbed, and distance travelled. The create function initiates a RobotStats object with all attributes set to zero. For each attribute, there is a set and get function to allow access and manipulation of the attribute values.

StatsTracker (Model)
The StatsTracker class is a collection of all the robot stats for each robot in the current game. Its sole attribute is an array that contains the RobotStats for each robot in the game. Its constructor function makes an object that takes in every robot in the games and creates a RobotStats for each one in the array.

VIEW

View (view)
This extends JFrame. The View class is the "view" in MCV architecture and is a frame that displays the various screens of the game. The createView function initiates the view frame at the desired size while the createPanel function creates a basic panel which contains no content. The next 7 functions display the various Panels of the game including the MainMenu, HelpPanel, GamePropertiesPanel, TeamSelectionPanel, WinnerPanel, StatsPanel, and GamePanel with an actionListener, a keyListener and the StatsTraacker, and GameInfoProvider implemented as required.

MainMenu extends JPanel
MainMenu class has only one function, its create function which initiates a MainMenu object with three Jbuttons: Begin, Help and Quit.

HelpPanel extends JPanel
The HelpPanel class only has a create function that makes an object with the two Jbuttons, Back and Quit as well as a block of text describing the rules of the game.

GamePropertiesPanel extends JPanel
The GameProperties has no attributes, and it’s create function generates 3 sets of radio buttons for the number of players, number of humans, and the board length.

TeamSelectionPanel extends JPanel
The TeamSelection panel applies a create function that adds the correct Jbuttons, text fields and sliders to the user interface. It also implements the RobotLibrarian which allows the uploading and downloading of stored robot teams as well as the creation of new ones, and the editing of old ones. The updatedisplayTeams function updates the panel to show any edited, added or removed teams.

WinnerPanel extends JPanel
The WinnerPanel has only a create function which constructs an object with three Jbuttons called Play Again, Stats, and Quit.

StatsPanel extends JPanel
The StatsPanel has no attributes and its only function is it’s create function, which builds an object with two Jbuttons, Play Again and Quit, and a display of the updated stats of the game.

GamePanel extends JPanel
The GamePanel is the actual game board panel on which the game is played. It has no Attributes and four functions. The GameCreate function creates the basic game panel that displays the game with the correct objects and starting statistics. The drawImage function

draws an image of a specific type at a specific location. The drawString function draws a string of characters at a specified location. The updateGame function updates the view by checking if the positions of the game objects have changed and calling the drawImage and drawString functions to redraw the game panel with the correct new positions.

Summary

In conclusion, we decided upon a Model-View-Controller architecture due to the advantages it has over standard architectures for our current project. It will allow us to create a more cohesive that is easier to work with and implement. Upon revaluation of our requirements document, we decided that we will not have time to include any of the "could have" features that were outlined in that document (such as animation and sound). Instead, we added an end turn button; basic individual robot statistics on the game screen; and functionality to prevent the robot from moving outside the arena. We believed that these additional functions are imperative to have a successful game. The classes were all designed to fit within either the view, controller, or model. All visual display classes are created under the view; all management classes that deal with input and interpretation are under the controller; and all other classes will be under the model. Using UML diagrams we were able to determine exactly how many classes interacted with each other within each level of architecture and how the architecture interacted as a whole. Using this information, we will be prepared to begin implementing the code necessary to develop the game.