Design Document
CMPT 370

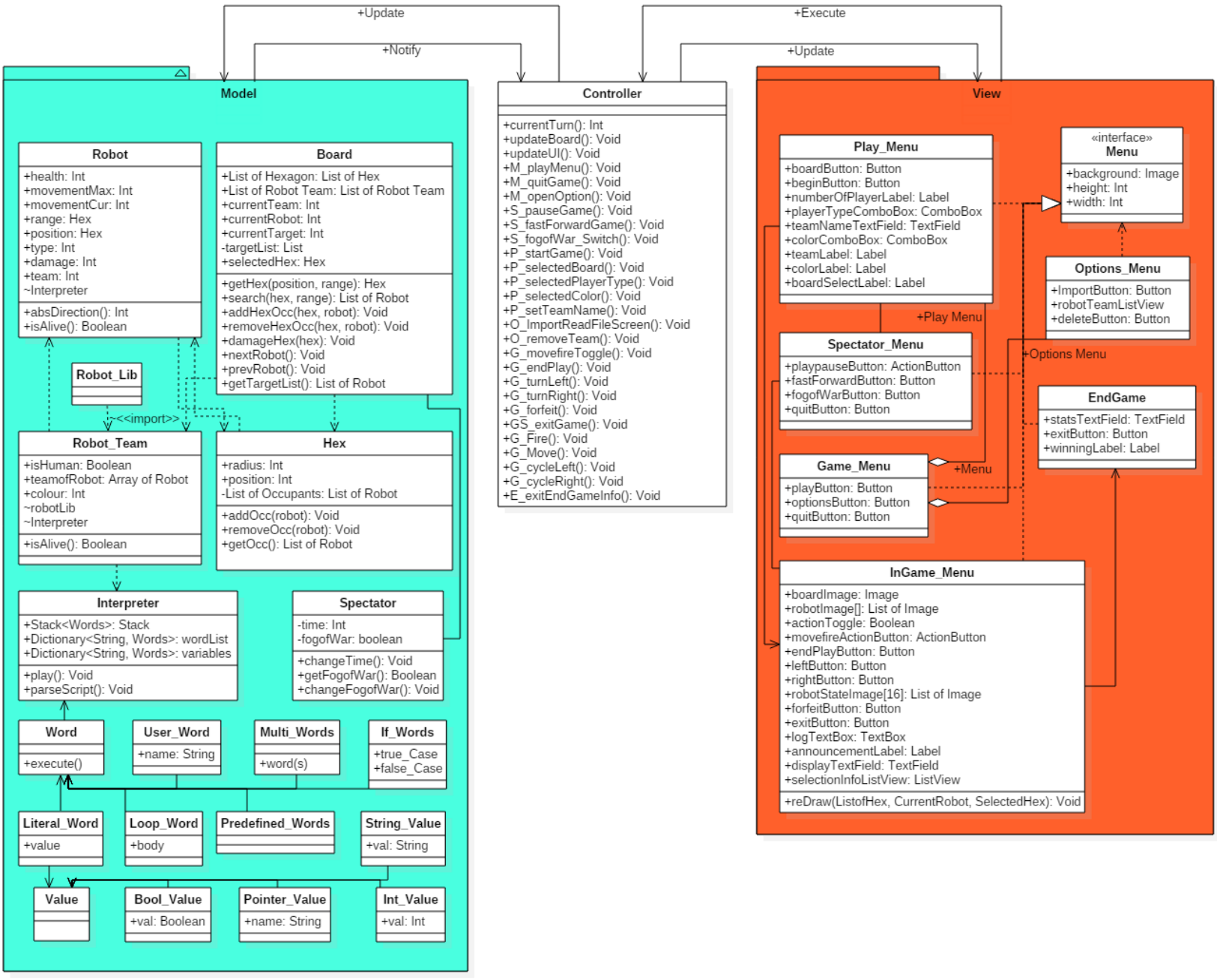
Group C4
Jack Huang
Brandon Jamieson
Ixabat Lahiji
Daniel Morris
Kevin Noonan

Architectural Choice

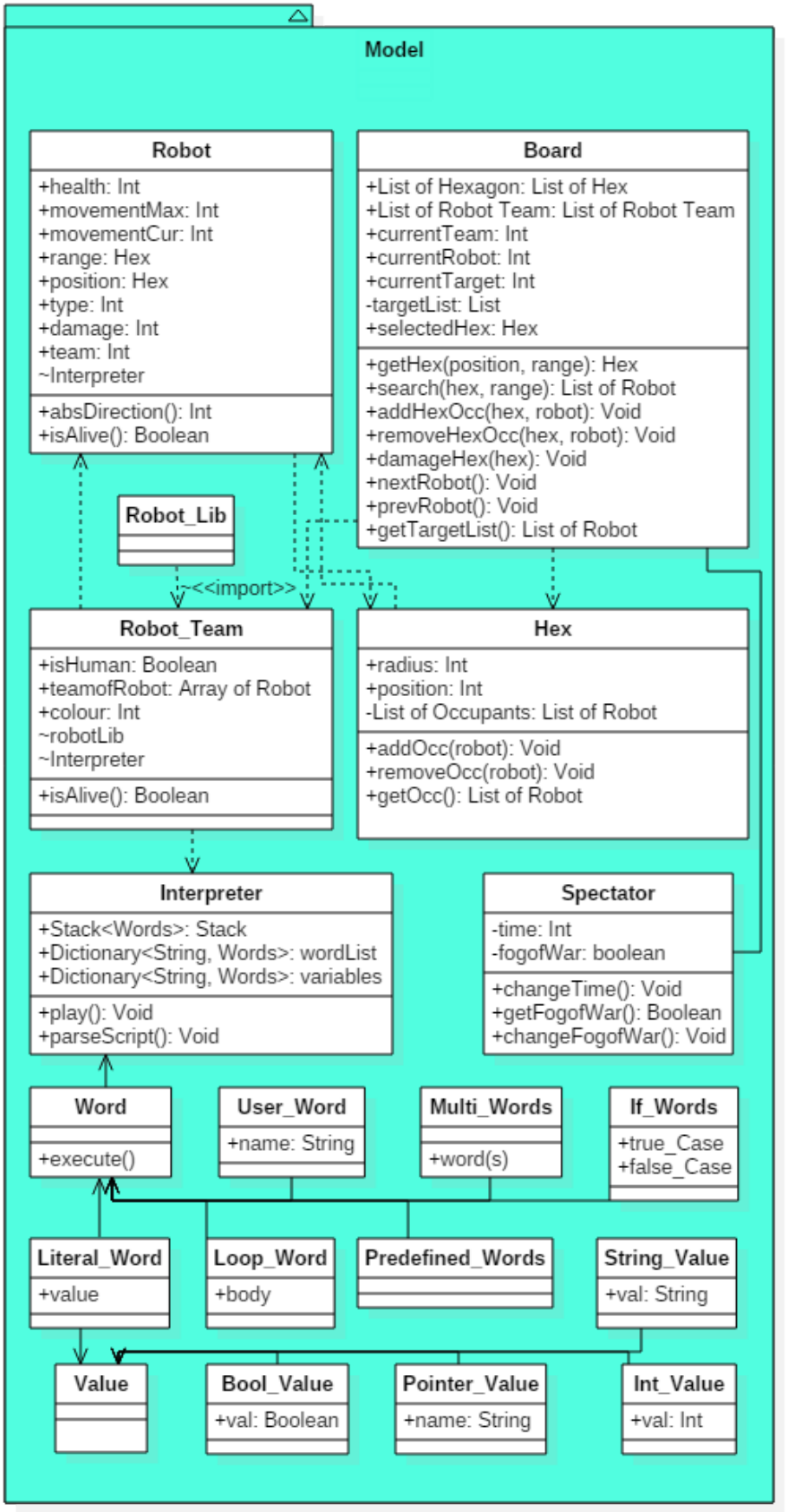
Our team's design makes use of the model-view-controller architecture. We chose this architecture because it plays to our team's strengths, as it is the architecture that we are most familiar and experienced with. One benefit of this architecture, is that model-view-controller cleanly separates two major components, the model and the view. This allows us to develop the two separately without relying on each other, as well as providing the user with a clear, bounded interface to the system. On top of this, the architecture keeps all robot-handling confined to a single major component: the model.

The view will be event-driven, handling interactions with users and relaying information to and from the controller. It will serve as an interface into the system and will inform the controller of user-input. The controller will also be event-driven, handling interactions with actors by manipulating the model. It will receive and handle user-input from the view, as well as updating and fetching from the model as needed. The model itself will have aspects of both the blackboard and database styles, due to some of the components inside. The robot librarian will serve as a database to be accessed by the controller for operations involving the access of robot teams. The game's board will act as the blackboard, with the controller manipulating the state of it as gameplay progresses.

One architectural option we considered instead of model-view-controller was a modified three-layer architecture. The top-layer would have been event-driven and handled GUI interaction. The middle-layer would have been blackboard style and the bottom-layer database style. Intermediate information would be stored in the middle-layer for game use, as well as receive data from the bottom-layer. We reasoned against this architecture because it did not encapsulate the guts of the game to one major component and instead, divided it among two of the layers. Model-view-controller provided a clear division between the guts and the rest of the system so we agreed upon it as our overall architecture.



Following the model-view-controller architecture, we have divided our system into those three major components. The view and model are independent of each other and only know about the controller. This keeps our system encapsulated, as well as expandable for the future. The controller will act as the system's intelligence and relay information between the model and the view, as well as updating the two of them.


Model
Robot
+health: Int
+movementMax: Int
+movementCur: Int
+range: Hex
+position: Hex
+type: Int
+damage: Int
+team: Int
~Interpreter
+absDirection(): Int
+isAlive(): Boolean
Board
+List of Hexagon: List of Hex
+List of Robot Team: List of Robot Team
+currentTeam: Int
+currentRobot: Int
+currentTarget: Int
-targetList: List
+selectedHex: Hex
+getHex(position, range): Hex
+search(hex, range): List of Robot
+addHexOcc(hex, robot): Void
+removeHexOcc(hex, robot): Void
+damageHex(hex): Void
+nextRobot(): Void
+prevRobot(): Void
+getTargetList(): List of Robot
Robot_Lib
~<<import>>
Robot_Team
+isHuman: Boolean
+teamofRobot: Array of Robot
+colour: Int
~robotLib
~Interpreter
+isAlive(): Boolean
Hex
+radius: Int
+position: Int
-List of Occupants: List of Robot
+addOcc(robot): Void
+removeOcc(robot): Void
+getOcc(): List of Robot
Interpreter
+Stack<Words>: Stack
+Dictionary<String, Words>: wordList
+Dictionary<String, Words>: variables
+play(): Void
+parseScript(): Void
Spectator
-time: Int
-fogofWar: boolean
+changeTime(): Void
+getFogofWar(): Boolean
+changeFogofWar(): Void
Word
+execute()
User_Word
+name: String
Multi_Words
+word(s)
If_Words
+true_Case
+false_Case
Literal_Word
+value
Loop_Word
+body
Predefined_Words
String_Value
+val: String
Value
Bool_Value
+val: Boolean
Pointer_Value
+name: String
Int_Value
+val: Int

The Model

The model will be one of the three major components in our system, and will house all the data and guts of the game. The controller will update the model based on input and gameplay, and the model will notify the controller to update the view. Since this is the case, the model implements several entities to keep track of the state of the game data-wise. Robots, teams, hexes, and the board itself are all contained within the view, among others.

Board

The board deals with managing moves and players. It will contain references to the robots on the board, references to the current team/robot in play, as well as positions and targeting.

- The getHex() method will return a Hex object, based on a distance and range that is passed into the method.
- The search() method will return a list of the Hexes that contain a robot on it, using a passed in Hex value as the origin, and moving outwards in a specified range.
- The addHex() method takes in a hex and a reference to a robot, and adds the referenced robot to the Hex.
- The removeHex() method takes in a hex and a reference to a robot, and removes the referenced robot from the Hex.
- The damageHex() method applies damage to the Hex value specified. Amount of damage applies is determined by the current robot's stats.
- The nextRobot() method changes the selected hex to the next robot that is available to be attacked by the current robot based on the current robot's range.
- The previousRobot() method changes the selected hex to the previous robot that is available to be attacked by the current robot based on the current robot's range.

The changes in the board will call an update to the view when necessary.

Hex

The hex class contains the info for a specific hexagon on the board. The radius and the position determine where the hexagon is on the board. The radius determines the amount of spaces away from the center of the board the hexagon is. The position determines which hexagon the specified hexagon is in that range starting from 0 to the right of the center then incrementing upwards clockwise. The list of occupants is a private variable in the hex will be a list of robots that are currently on that hex's position coordinates.

- The addOcc(robot) method will add the robot to the list of occupants

- The removeOcc(robot) method will remove the robot from the list of occupants
- The getOcc() method will return a list of all the robots on that hex

Robot Team

The robot team class is used to group robots together into one team. The isHuman field determines if the robot team being controlled is human. The teamOfRobot field is an array of three robots associated with that team. The colour field is an int, representing what colour the specified team is.

- The isAlive() method will check all three of the robots and if at least one of them is alive then the method will return true.

Robot

The robot class contains the statistics about each individual robot on the robot team. The health keeps the health amount the robot has or has left. movementMax and movementCur fields keep track of the number of possible moves the robot has and the moves that it has left, respectively. The range field keeps track of how far the robot can shoot, in terms of the furthest hexagonal board piece. The position field keeps track of the hexagonal board piece that the robot is currently placed on. The type field is the robot's type which can be scout, sniper or tank. The damage field is the statistic for the different type of robot's damage that it can output when attacking. The team field is an int that represents the team the robot is currently on.

- The absDirection() method returns an integer between 0 and 5 that displays the current direction the robot is facing.
- The isAlive() method returns a boolean that displays whether the robot is alive or if it is dead, based on the value of the health field.

Spectator

The spectator class contains the required information for when the spectator is active. The spectator would become active when there are no longer any human teams alive. The time field is an integer, either 1, 2, or 4. This value determines the length of time in between plays. In between plays that the spectator is watching, the delay time is the calculated by the default time divided by the time variable. The fogofWar boolean field will determine whether or not the UI displays the fog of war for the current robot's vision range.

- The changeFogofWar() method will change the fogofWar field.
- The changeTime() method will double the time field unless the time field is currently 4 then the time field will be set back to 1.
- The getFogofWar() method will return the fogofWar field.

Interpreter

The robot interpreter will handle all the *RobotSport370* interpretation for the robots. It contains a stack for the robots to push and pop values onto and from by executing words. It will contain two dictionaries, both of which will use strings as keys. The first will be a word list containing all system-defined words. The second will be a variable list containing all user-defined words. These will be used when parsing scripts for the robot. With each robot team having an interpreter, our system will be able to interact with the robots and give them operations at ease.

- The play() function will be called to tell the correct robot it is their turn to play.
- The parseScript() function will be called to parse a script and populate the word list and variable list.

Word

Robots will perform their operations by executing either a system and user-defined "word". All words will contain an execute() function that will vary depending on the type of word. Words may be literal words, system-defined words, or user-defined words. Literal words will push values to the stack (boolean, int, or string), while defined words will perform operations on the robot when executed. Several subclasses of words are defined to support these operations.

User word

- "User" words are defined when a script is parsed. They will have a string containing the user-defined name and will go into the variables list. All other words however, will go into the word list.

Multi word

- "Multi" words simply contain a list of words that may be accessed to execute. This way sequences of words can be performed easily.

If word

- "If" words contain both a true case and a false case for performing evaluations. This would include operations like "<" or !="

Predefined word

- "Predefined" words" are words defined by the system on initialization (such as "shoot!" or "turn!").

Loop word

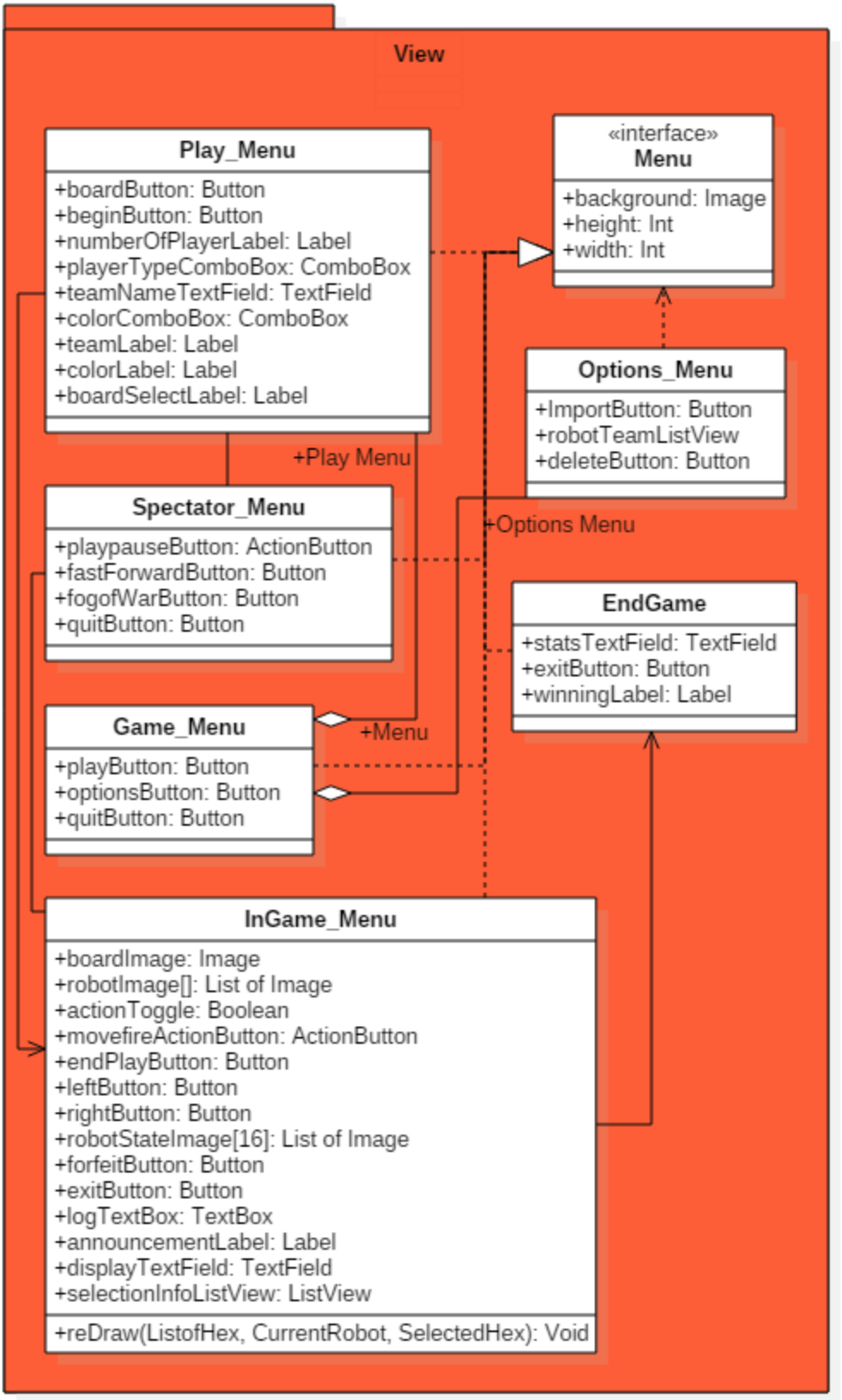
- "Loop" words contain a body of operations to perform. They pop off the end and start of the loop and execute the body until the end is reached.

Literal Word

- “Literal” words defined by the system and contain a value that is simply pushed onto the stack when executed.

Values

Values may be one of the following types: **Boolean**, **string**, **int**, or **pointer**. When pushed onto the stack these values may be used by system and user defined words to manipulate the robot. While the first three represent themselves, pointer values will simply contain a string of the pointer value.


View
Play_Menu
+boardButton: Button
+beginButton: Button
+numberOfPlayerLabel: Label
+playerTypeComboBox: ComboBox
+teamNameTextField: TextField
+colorComboBox: ComboBox
+teamLabel: Label
+colorLabel: Label
+boardSelectLabel: Label
«interface»
Menu
+background: Image
+height: Int
+width: Int
Options_Menu
+ImportButton: Button
+robotTeamListView
+deleteButton: Button
+Play Menu
+Options Menu
Spectator_Menu
+playpauseButton: ActionButton
+fastForwardButton: Button
+fogofWarButton: Button
+quitButton: Button
EndGame
+statsTextField: TextField
+exitButton: Button
+winningLabel: Label
Game_Menu
+playButton: Button
+optionsButton: Button
+quitButton: Button
+Menu
InGame_Menu
+boardImage: Image
+robotImage[]: List of Image
+actionToggle: Boolean
+movefireActionButton: ActionButton
+endPlayButton: Button
+leftButton: Button
+rightButton: Button
+robotStateImage[16]: List of Image
+forfeitButton: Button
+exitButton: Button
+logTextBox: TextBox
+announcementLabel: Label
+displayTextField: TextField
+selectionInfoListView: ListView
+reDraw(ListofHex, CurrentRobot, SelectedHex): Void

The View

On the other end of the controller, we have the view, which will serve as the interface to the whole system. Users will be able to interact with the view which will notify the controller of their input and manipulate the game from there. The view will also be updated as the game progresses, with the controller feeding it information from the model. Our view is composed of several GUI components with different interactions depending on the game state.

Game_Menu

The game menu is the main menu of the game, the will be what the user see first when the game is launched. The game menu allow the player to navigate between play and option menu and the ability to quit the game.

Play_Menu

The play menu is the first menu that the player will see when they open the game. This menu will be located inside of the Game_Menu and will allow the player to set up a game to play. The menu will take in input for number of teams, team names, team types, team colours, and board size. The available board sizes will depend on the number of teams. Once the begin button is pressed, the menu calls the controller's startGame() method.

Spectator_Menu

The spectator menu appears when the last human player dies, or there is no human player in the game to begin with. The spectator menu allow the player to control the game in spectator mode. The user is given play and pause, fast forward toggle, fog of war toggle, and quit button. Fastforwarding will cycle through 1x, 2x, and 4x, to allow for a change of game pace when spectating. The fog of war toggle will decide whether fog of war is to be hidden or displayed based on the current team playing. The quit button will remove the user from the game and place them at the main menu.

InGame_Menu

The in-game menu is the GUI screen that is viewed for the actual playing of the game. This is composed of many buttons and displays for the player to use and view. The boardImage field holds the hexagonal playing board image, where the game takes place. The robotImage field holds a list of each individual robot's image that will be viewed on the game board. The announcementLabel will inform the player if it is their turn, if they are waiting to have a turn, or if they are spectating. The GUI will also include a log of the game's plays for testing purposes.

The toggleSwitch field keeps track of whether the player is shooting or moving the current robot. The player will have a button for performing the current action, either moving or shooting, and will have a left button on one side and right on the other for cycling the selection. These buttons are stored in the fields leftButton and rightButton. When moving, the selection will rotate the current robot clockwise or counterclockwise one direction. When attacking, the buttons will cycle through the available target hexes to shoot. To end the play the player will click on the end play button which is stored in the endPlay button field. The selection info box will appear when hexes are cycled over and will display the occupants and their stats. The fields exitButton and forfeitButton are used to quit the in game screen and to remove the player's team from the game early. The robotStateImage field is a list of the robots info as the game goes on it keeps track of the robot's attack, movement, health and team.

- The reDraw method will redraw the board when called based on the controller specifications.

EndGame

The EndGame screen will display the winner of the game, while also showing all the information that happend during the game for each team, like damage inflicted, damage absorbed, amount survived, amount destroyed, distance travelled, shots fired. All the data will also be sent to the server to update the stats of all AI that played in the game.

Options_Menu

The option menu allow the user to add and remove a robot team from the game. Robot teams are added into the game by importing from the local JSON files, user can remove by clicking the remove button. The option menu would also display all the robot team that is already inside of the game in a list, with informations like team name, wins/losses and match played.

Menu

The menu will serve as an interface to all GUI menu elements. It will contain a background image, as well as a height and a width. Having all other menu elements implement this interface will ensure that they all are to the specified details visually and proportionally.

The Controller

While the view simply handles interaction with the user, the controller will serve as the intelligence of the system. It will have access to both the model and the view and will send/receive updates to and from the other major components. User input will come from the view and notify the controller of what action to take next when required. The controller will update the model of changes and the model will notify the controller to update the view based on such changes. In this way, we can keep all our game flow and intelligence bound to a single component, making modification and expansion easier.

To serve its purpose, the controller implements a plethora of methods for various interactions. Methods have been sorted into six categories pertaining to their interaction type: Play Menu (P), Spectator (S), In-Game (G), End-Game (E), Options (O), and Main Menu (M).

- The currentTurn() method will return the integer value of the current turn.

- The updateBoard() method will call the reDraw() method of the InGame_Menu entity to redraw the board to present it accurately with the given parameters.
- The updateUI() method will tell the view to update certain UI elements in game, such as displaying robot movement or health.
- The M_playMenu() method will display the play menu where the user will select the number of players, board size, and other options.
- The M_quitGame() method will exit the user from the game.
- The M_openOption() method will open the options menu where robot importing will be handled.

When there are only AI players in-game, no humans, there will be options for manipulating the flow of the game. The spectator will be able to invoke the following functions through the view.

- The S_pauseGame() method will pause the game until called again.
- The S_fastForwardGame() method will fast forward the speed of progression of the game by a multiplier.
- The S_fogofWar_Switch() method will toggle whether fog of war is displayed or not.

The play menu will be displayed when the user presses “Play” on the game menu. It will prompt the user for a board size, player types (Human or AI), and their team name and color. The player will then be able to begin the game by pressing a button.

- The P_startGame() method will start the game with the user-inputted parameters.
- The P_selectedBoard() method will change the currently selected board size, depending on user-input and player amount.
- The P_selectedPlayerType() method will select the player type of a certain team. This can be human controlled or AI controlled.
- The P_selectedColor() method will select the color of a team.
- The P_setTeamName() method set the name of a team.

The options menu will be displayed when the user presses “Options” on the game menu. It will have functionality for importing teams as well as removing teams. This way we can manipulate our robot library through the user interface.

- The O_ImportReadFileScreen() method imports a robot team to the system to be displayed on the view and used in-game.

- The O_removeTeam() method removes a robot team from the system and updates the view as such.

Once the user has started a game, assuming there are human players in the game, they will be able to give their tanks commands via buttons. The view will notify the controller of these inputs and the controller will use an assortment of methods to manipulate the current robot and game state.

- The G_movefireToggle() method will toggle between moving and firing mode, and will update the view to display options for the respective action.
- The G_endPlay() method will end the current robot's play and begin the next robot's play.
- The G_turnLeft() method will rotate the current robot once in the counterclockwise direction.
- The G_turnRight() method will rotate the current robot once in the clockwise direction.
- The G_forfeit() method will forfeit the game for the current human's team, removing them from the game and moving them to spectator mode if there are no other human players.
- The GS_exitGame() method will exit the game, and can be fired when the player is in spectator or during the game.
- The G_Fire() method will be toggled if the player clicks the movefireActionButton while the player is in shoot mode. This method will deduct the current robot's damage from all of the robot's health that are on the selectedHex.
- The G_Move() method will be toggled if the player clicks the movefireActionButton while the player is in move mode. This method will change the position and/or range of the robot based on the direction of the robot.
- The G_cycleLeft() method will first check if the player is in move mode or shoot mode. If the player is in move mode then the function will change the current robot's rotation once to the left. If the player is in shoot mode then the selectedHex in the board class will be changed to the previous available target to shoot.
- Similarly, the G_cycleRight() method will first check if the player is in move mode or shoot mode. If the player is in move mode then the function will change the current robot's rotation once to the right. If the player is in shoot mode then the selectedHex in the board class will be changed to the next available target to shoot.
- The E_exitEndGameInfo() method will change the current view to the Game_Menu where the player will be able to start a new game.

Changes from our Requirements document and planning

We made several tentative specifications in our requirements document that we modified in our design document. Things we had previous classified as "should have" or "nice-to-have" included: operations such as clicking hexagons to move or attack, as well as having tanks visually stack on the same hex.

After much discussion, we decided that clicking hexagons to move and attack was not a feasible option for our design. Due to the nature of the robots and their individual orientations and relative coordinates, making click-based movement would be hard to implement. It would require some form of translation to an "absolute" board coordinate independent of the robot's relative coordinates, and then moving the robot based on this. We would also have to include functionality for if the user clicked multiple hexes away from the current robot, involving some sort of path-finding.

To avoid all these issues, we decided that movement and attacking could be performed via buttons. When moving, the user will rotate their tank using left and right buttons and move in the current direction by pressing "Move". When attacking, the user will use the left and right buttons to cycle through potential hexes to shoot at, and will fire on the currently selected hex by pressing "Fire".

The notion of having tanks visually stack on the same hex was also modified in our design. Originally, tanks would stack and health bars would be displayed above them. We realized this may be unclear visually for the user, as health bars would end up stacked as well. Implementing some way of aligning health bars to avoid overlap while still providing clarity as to which robot they belong to would be a tough task to fulfill.

To combat this issue, our team decided to implement a "Hex Info" display panel, corresponding to the selection info field of the view in-game. This panel will display the list of robots occupying the selected hex, as well as their health. This way, we can clearly notify the user of the tanks on any given hex without anything visually interfering with the game board itself. On top of this, we can simply update the info of one panel as opposed to updating the alignment of individual tanks. This also removes our need for a health bar for each robot.