Robo-Wars Design Document

Team D4

Nickolas Gough, Scott Hebert, Janelle Hindman, Yige Huang, and Tushita Patel

Project: Robo-Wars

Dr. Christopher Dutchyn

CMPT 370 Term I 2016/2017

Executive Summary

Robo-Wars design has two major levels: a high-level architecture level, and a lower-level detailed design. In terms of architecture, it has a mixed Independent Components (IC) with some level of Model-View-Controller (MVC) as well. The original choice was IC for greatest flexibility in design, but if you group the components in a certain way, an MVC architecture can be attained as well. This can assist in reasoning about the design if an MVC architecture is more familiar.

The six (6) components identified for the design are the Flow component, the Display component, the Board component, the Pieces component, the Interpreter component, and the Logger component. In terms of a detailed design, each component is decomposed into multiple Classes by major area of responsibility. Most components have between three (3) and five (5) Classes.

As part of this Design process, two (2) parts of the Requirements document were revised. Much of the Robot Librarian Actor actions have been removed due to duplication with internal actions, and the Zooming function has been moved to a nice-to-have status given its technical difficulty relative to its gain.

Contents

1.0 Proposed Architecture

The game architecture is an architecture of Independent Components (IC). The major components in the architecture are the Logger, the Pieces, the Board, the Display, the Interpreter, and the Event Handler components. A justification for the architecture will be discussed first, and then the components will be discussed in further detail.

1.1 Architecture Justification

The IC architecture was chosen for the game for several reasons, among them being minimal dependency among game entities, the event-driven nature of the game, and the familiarity of the design team with the architecture. Each of these reasons will be discussed in terms of their strengths and risks, and then alternative architectures will be discussed and why they were not chosen.

1.1.1 Minimization of Dependency

The most important reason for selecting an IC architecture is that it imposes the minimum dependency among the game entities. This has several benefits. One benefit is that parallelization of class and code construction can be maximized, an important benefit in group-oriented software challenges. Another benefit is modularity of testing and deployment. If the game is effectively decomposed, then testing on a unit level rather than a system level can be maximized, allowing for more controlled and rapid testing procedures.

A risk associated with the minimization of dependency, however, would be the lack of structure. The IC architecture has some of the least structure of any architecture, and as such needs to be carefully managed to prevent issues in later phases. While this risk is noted, the requirements document is quite detailed in the Actor/System interactions, which will provide support here. Like a house with many windows, the detailed nature of the requirements document provides more insight into the game's inner workings and illuminates the structure.

1.1.2 Event-Driven Nature of the Game

Another important reason for selecting IC as the game architecture is that IC architectures effectively works with event-driven systems, and the game is highly event-driven. The IC architecture models

events explicitly inside the game system, which makes it easy to use the same framework to think about the system and how it interacts among itself and also with Actors outside the system. That the entire human-game interface relies on events to work properly is a strong secondary line of reasoning here.

The major risk that relying on the IC architecture exposes with event-driven games is clear understanding and elicitation of the events. Should major events not be included in the design, entire communication pathways can be left undersigned, causing problems in the code construction phase. This risk is mitigated for the game partly because of the nature of general game structure. The possible events have been well documented and in these cases, the likelihood of missing major events should be minimal.

1.1.3 Familiarity of Design Team with IC

A final important reason for using the IC architecture is that the design team is already very familiar with this architecture. This has the major benefit of allowing the design team to focus its efforts on areas where greater clarity is necessary such as the component analysis and class diagram elucidation. If a less familiar architecture were selected over IC, this would dilute the design team's concentration and could easily result in a poorer design.

Of course, the risk of familiarity is that an incorrect or subpar choice is made solely due to familiarity. If this occurs, then in the worst case the entire design would have to be redone. However, in this case, it is well-established that games work well with an IC architecture, and this is not the sole or even most important reason for using it.

1.1.4 Another Architecture - MVC

An IC architecture is hardly the only applicable architecture that could have been chosen for the game. Another popular architecture is the Model-View-Controller (MVC) architecture. The strength of the MVC architecture relative to the IC architecture is that the assignment of responsibilities is typically much clearer in the MVC architecture. However, this is due to the greater structure provided by the MVC architecture compared to the IC architecture. Should this structure not add value to the design, the design suffers from becoming more rigid than necessary. Another issue with MVC is that the division of testing and code construction responsibilities can be more difficult. Mainly for this latter reason, MVC was not chosen as the primary game architecture. However, as seen below, some components when combined could be seen to provide some of the functionality of the MVC architecture.

1.2 Component Details

Now that the architecture choice has been justified, the components can be discussed in detail. The components will be discussed primarily in terms of their responsibilities at a large scale, as well as the information flow among the components. This naturally leads to the detailed design discussion in the next section.

1.2.1 The Display Component

The Display component is responsible for the user interface (UI) elements of the game. It primarily takes information from the Board and displays that information to the user. The Display along with the Flow would be the View component of an MVC architecture. Even more than the Pieces component which appears next, the Display is a passive component and does not send information out to other components.

1.2.2 The Flow Component

The Flow component holds the responsibility for gathering the Player input—using mouse clicks—and forwarding it to the Board if it had gameplay ramifications, and to the Display if it did not. There is no expectation of communication to the Event Handler from any component.

Again, from an MVC perspective, the Event Handler could be partly View in that it interacts with the outside users of the system, but it can also be partly Controller, as it has to know where to send events. The other major View-like component in the system is the Display component.

1.2.3 The Board Component

The central component of the architecture for the actual play of the game is the Board component. It is responsible for maintaining the overall board state and communicating that state to the other components. An example would be to tell the Display when something needs to be updated. Another would be to request information from the Pieces regarding their status (as this is needed to maintain the board state). It is expected that most other components communicate with this component. Looked at from an MVC architectural perspective, the Board has some of the functions of a Controller, but also some aspects of the Model. One of the primary inputs to the Board component comes from the Event Handler component.

1.2.4 The Pieces Component

The Pieces component is responsible for storing and tracking information relevant to the pieces of the game. It is not, however, responsible for knowing the location of the pieces; that is handled by the Board component. There is a great deal of communication between the Pieces and the Board. The Board receives the commands from either the Interpreter or the Event Handler and then adjudicates them based on information obtained from the Pieces component.

1.2.5 The Interpreter Component

As mentioned above, the Interpreter is one of the components that sends information to the Board. As its name suggests, the Interpreter's main responsibility is to interpret the scripts for the computer player or players and send that information to the Board for processing. It can be seen as the AI analog to the Event Handler component as it is the primary source of input into the system for computer players in the same way that the Event Handler component is the primary source of input into the system for human players.

1.2.6 The Logger Component

The final component to the architecture is the Logger component. Its responsibility is to store every action that occurs in the game. It also can send that log to the display when requested along with a given context. For example, if a team asks to view the log, the log displayed should only show the information that that team performed or saw performed.

1.2.7 The Complete Architecture

These six components comprise the system architecture. This architecture is summarized in the figure below, along with the important information flows.

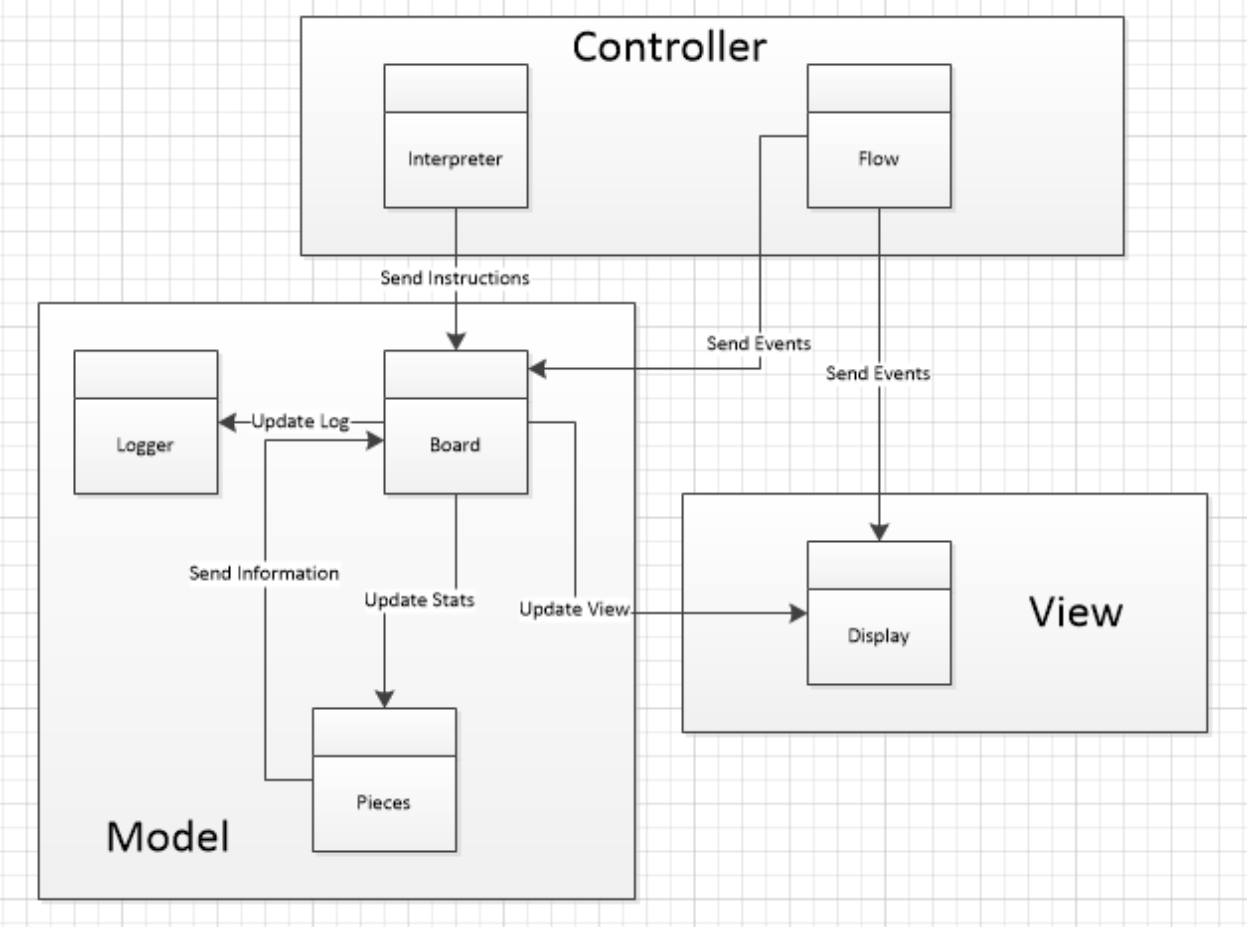

Figure 1: IC/MVC Architecture - High Level View

2.0 Detailed Design

Once the Architecture is complete, Detailed Design can begin. This consists of decomposing each component into one or more Classes, and determining the data elements (fields) and actions (methods) necessary to carry out the responsibilities of that component. The Classes will be organized by the component to which they belong.

2.1 Display Component Classes

The Display Classes handle the actual interfaces that the player of the game uses to play the game. There is a main Display Class, a Screen Class, and a subclass of Screen, GameScreen, that handles the major aspects of the interface. First is the Display Class.

2.1.1 Display Class

The Display class is designed to model the display component. The class stores an array of all the screens, which can be easily indexed using the ScreenEnum enumeration that is assigned to each of the screens of the Screen class. The public switchToScreen() method takes in the enumeration of the screen to switch to and then updates the component to display the specified screen. The private initializer methods are used to initialize the various screens held by the display. The display also provides an accessor to retrieve the game screen to allow easier updating of the game screen.

The Display class contains the following fields:

- Screen[] screens: The screens field is an array of Screen elements (see Section 2.1.2). It contains all of the screens in the game which will be displayed, indexed by the value of the ScreenEnum (defined below).
- ScreenEnum currentScreen: The currentScreen stores an enumeration value representing the current screen being displayed.

The fields in the Display class depend on the definition of the ScreenEnum. The ScreenEnum is an enumeration, containing a fixed set of values which map to all of the screens in the game. It will contain the values TITLE, PLAYERSELECTION, GAMEOPTIONS, TURNTRANSITION, GAME, RESULT, ROBOTARCHIVE and SETTINGS.

The Display class is responsible for initializing all of the screens in the game, which will be displayed to the user. Since it is the first component to be initialized, and since its screens contain many interactive elements like buttons, it is also responsible for initializing the pieces of the Flow component, which depend on and must be linked to the buttons. It contains the following methods for initializing each game screen: initializeTitle(), initializePlayerSelection(), initializeGameOptions(), initializeTurnTransition(), initializeGame(), initializeResult(), initializeRobotArchive(), initializeSettings(). It also contains accessor methods for each of the screens that have been initialized: getTitleScreen(), getPlayerSelectionScreen(), getGameOptionsScreen(), getTurnTransitionScreen(), getGameScreen(), getResultScreen(), getRobotArchiveScreen() and getSettingsScreen(). Once all of the screens have been initialized, it will be able to use initializeFlow() to create the elements of the Flow component, which depend on the interactive elements within each screen.

The most important method of the Display class is the switchToScreen() method, which takes a ScreenEnum value as an argument. It will perform the operations necessary to switch to the screen specified by the value of the ScreenEnum, which include changing which screen is being displayed, and playing any necessary animations or transitions. switchToScreen() will not return a value, but will modify the value of currentScreen.

2.1.2 Screen Class

The Screen class has been designed to model a screen within the system. Each screen will have a specific layout, but the layout will be initialized by the display component. Each screen will be assigned an enumeration that will be stored and for which there is an accessor. Each screen will use the Swing library for its display components, and will contain a root JPanel which holds all of the display elements for that screen, including buttons, images, and other control widgets.

The Screen class contains the following fields:

- ScreenEnum screenType: screenType contains an enumeration value describing which screen is being displayed, as described above.
- JPanel base: This field holds the base JPanel for the given screen, which may have zero or more elements as its children. Together they comprise all of the visual elements of a given screen displayed to the user, including components that the user may interact with.

The Screen class's only methods are accessors for its two fields. It contains getscreenType(), which returns the ScreenEnum value, and getBase(), which returns the JPanel of the screen.

The Screen class is intended to hold the display elements of each screen. For all screens except the main game screen, the built-in methods for drawing these elements to the application window is sufficient. For the main game screen, additional methods and fields are required in order to depict the elements of the game properly, and so it must extend the Screen class to support this extra functionality.

2.1.3 GameScreen Class

The GameScreen class inherits from the Screen class and has been designed to model the game screen. It contains all elements that a Screen does, including its ScreenEnum value and the Panels it contains. A collection of images for the game pieces will be stored in an array that will be indexed using enumerations where applicable. The x and y offsets of the board will be stored to allow the correct portion of the board to display when the Player pans the board. Also, the log screen and the context menu that will be shown to the Player to allow the Player to select their action are be stored.

The GameScreen class contains the following fields:

- ImageIcon[18][6][] pieceImageLibrary: pieceImageLibrary is a 3D array of ImageIcons, which contains the various frames for the pieces which need to be drawn to the board. The first index of the array represents the type of piece, by colour and then type, where ImageIcon[0] contains the Red Scout images, and ImageIcon[17] contains the Purple Tank images. The second index of the array represents the rotation of the image, in six different directions. The third index of the array contains the various frames of animation that each piece might have (stationary, moving, shooting, dying).
- ImageIcon[] pieceImages: The pieceImages array contains references to the ImageIcon which each piece on the board is currently displaying. The size of pieceImages depends on the number of players and number of pieces assigned to each player.
- Integer boardOffsetX and Integer boardOffsetY: Both boardOffsetX and boardOffsetY represent the pixel offset coordinates for displaying the board, where (0,0) is the top-left-hand corner of the board display. These values are changed when the player pans the board.
- JPanel logWindow: This field holds the window containing the display representation of the entries in the Log for the current player (see Section 2.6).
- JPanel contextMenu: This field holds th context menu that will appear to allow the player to move a piece to a space or to shoot a space
- Logger logger: The logger field stores a reference to the logger from the Logger component (see Section 2.6) in order to access it for display.

GameScreen contains a method for initializing the main game screen and its subordinate elements, as described in the requirements document: initializeGameScreen(), which contains initializeLogWindow(), initializeSidePanel(), initializePlayerBar() and initializeContextMenu().

The main game screen contains many images which need to be drawn in a specific and dynamically changing order (such as board hexagons and pieces), so the GameScreen class contains an update()

method which allows the display to be redrawn to correctly represent the board. The update() function is intended to be called on every display refresh.

The update() method has several helper methods, which can also be called individually if necessary. movePiece() will take the index of the piece and its new position as a valid HexCoord (see SectionXXX), update the location of a piece in the display, and also update its frame in pieceImages if necessary (if the piece rotated, for example). shootSpace() will take a valid HexCoord and will update the image of the space on the board. The pan() method takes two integers, deltaX and deltaY, updates the boardOffsetX and boardOffsetY fields, and redraws the board to the screen based on the new offset. If the player has the Log window displayed, update() may also need to call updateLog(), which takes a TeamEnum value (see SectionxXX) and updates the logWindow field to the correct representation for a given team.

GameScreen also contains several methods which deal with showing and hiding various elements of the main game screen. The promptQuit() and promptEndTurn() methods will display the appropriate dialog windows that ask the player for confirmation of quitting or ending a turn. showContextMenu() and hideContextMenu() will cause the context menu for moving and shooting a space to display or be hidden.

Finally, the switchPlayer() method takes a TeamEnum value representing the current team and an integer representing the current piece, and updates the display at the beginning of a turn to focus on this team and piece. The display will be centered on the current piece, which may overwrite the values of boardOffsetX and boardOffsetY. The player bar displaying the status of each team colour must be updated as well.

2.1.4 Display Class Diagram

All of this is summarized in the Class Diagram shown below.

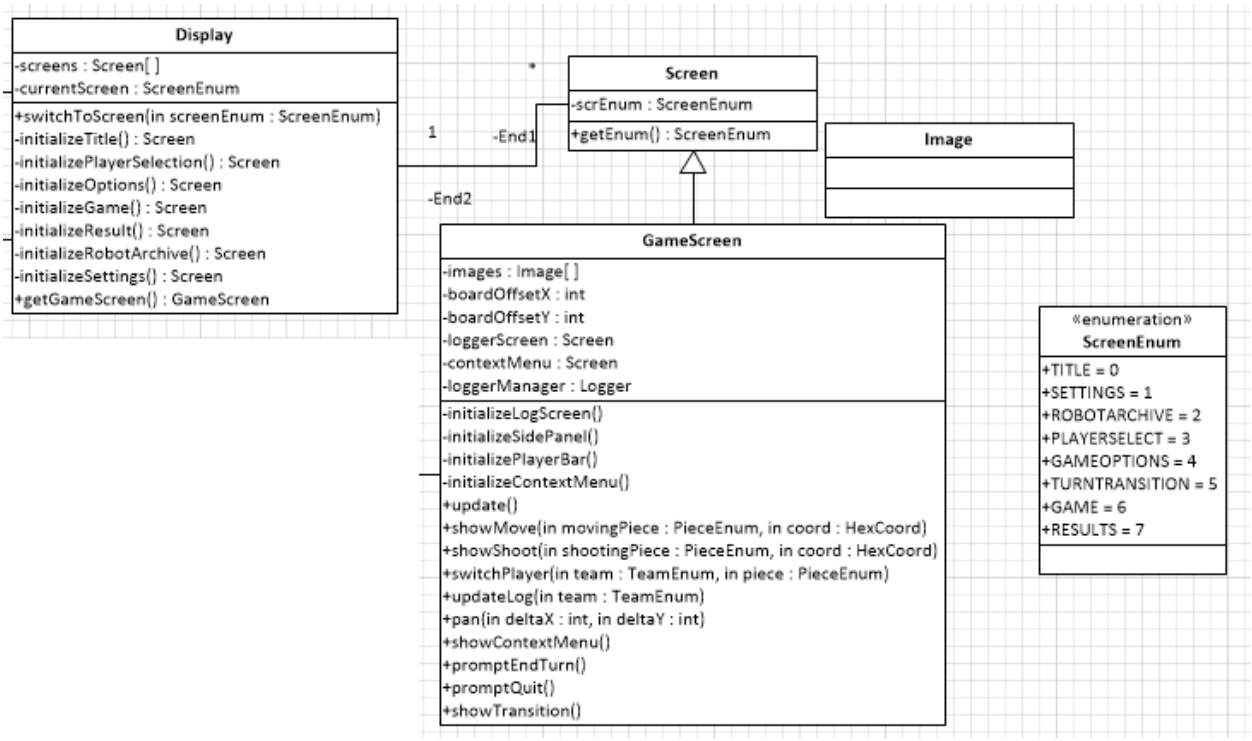

Figure 2: Display Class Diagram

Next after the Display Component Classes is the Flow Component Classes, which handle the interaction interface the same way that the Display Component Classes handles the visual interface.

2.2 Flow Component Classes

The Classes that comprise the Flow Component Classes primarily deal with either detection of interaction events (e.g., mouse clicks) or the menus that drive the creation of the game.

2.2.1 EventCatcher Class

The EventCatcher class, which implements the AWT Event Listener interface responds to various events that the User and the Player initiate. These events could be a click on the screen, a click on a button, or a key press on the keyboard. The Swing toolkit is used to handle these possible events, which is located in the Javax package.

The EventCatcher class contains the eventDispatched(AWTEvent) method, which is the hub of all events and it is located in the main EventCatcher class. It calls the delegate methods to perform the specific job based on the particular event. This method accepts the AWTEvent as a parameter.

2.2.2 Settings Class

The Settings class stores the information that the user would like to store about the future match. For example, the volume of the background sound during the match, whether or not the logs should be stored to the disk and additional settings such as keyboard inputs that the user would like to use for movement and direction of the pieces.

The Settings class contains the following fields:

- The Integer soundVolume holds a value for the loudness of the background sound on a scale from 1 to 100.
- The Boolean saveLogsToDisk determines whether the logs will be stored to the disk for future reference.
- The Dictionary<String, Integer> keyMapping stores the mapping of the keyboard key inputs to their functionalities.

The methods in the Settings class are the accessor methods and the mutator methods for the fields.

2.2.3 PlayerSettings Class

The PlayerSettings class contains information about the preferences of the player while playing the game. This class contains the following fields:

- The Boolean isAI denotes if the player is an AI or a human player.
- The String name is the name given to the player.
- The Dictionary<String, RobotRecord> mapRobots maps the name of the robot to the record of the robot.

The methods in this class are the accessor and mutator methods for each field. As an alternative to the setter for the dictionary, mapRobots, the method setProgram(String , RobotRecord) sets the AI piece identified by ID to a RobotRecord.

2.2.4 MatchOptions Class

The MatchOptions class is essentially for storing all the details about a match, if in case the player wants to rematch. The fields in this class include:

- The PlayerSettings[] players is an array of(maximum six) elements which stores the information about the player provided in the PlayerSettings class.
- The Integer numPlayers stores the number of players in the match.
- The String mapName is the name of the map assigned to the match.
- The Integer numScouts, numSnipers, and numTanks store the number of scouts, snipers and Tanks of all players in the match.
- The RobotRecord defaultScout, defaultSniper, and defaultTank store the record of the default Scout, Sniper and Tank, if the user chooses not to specify the Robot.

The methods in this class are the accessor and mutator methods for each of the fields.

2.2.5 RobotRecord Class

The RobotRecord class has the sole purpose of storing the JSON file.

The class contains only one field:

- The String [] JSONFile stores the JSON input as String in the form of an array.

2.2.6 MenuManager Class

The MenuManager class helps with the user inputs given on all screens that are interacted by the user. In other words, all events in all the screens except for the game screen are reported to the MenuManager class. As a result, the MenuManager class stores information about the match options and the general settings.

The class contains the following fields:

- The Settings settings stores the preferences from the settings
- The MatchOptions matchOptions is the field which contains all the information about the match given in the MatchOptions class.
- The List<RobotRecord> robotRecords stores a list of all records of the robots in the game.
- The String [] buttons stores all the possible button names in their scope of screens.

The method clickedButton(String) is located both in MenuManager class and GameManager class. The method in the MenuManager deals with the buttons clicked on all screens other than the game screen, and it takes in the name of the button which is clicked as a parameter, throws an exception if the button

name is invalid, calls the Display component to switch between the screens as requested or executes the expected request and returns nothing.

A private method IsValidButton() is used to check whether or not any of the arguments of button names are valid. The method loadSettings(), as the name suggests, loads settings from the file, or creates a new file if none exists. The method neither returns anything nor accepts any parameters. Similarly, the method saveSettings() updates the values of the settings and saves to file. Likewise, the method neither returns anything nor accepts any parameters.

2.2.7 GameManager Class

The GameManager class handles all the interactions between the player and the screen.

It consists of the following fields:

- The MatchOptions m stores the information about the match.
- The String[] buttons contains all possible button names in the game screen.

The class contains a method, createMatch(MatchOptions), which assigns the match options to the match field.

The method clickedButton(String) is also located in the GameManager class. This method in the GameManager handles the buttons on the game screen. It takes in the name of the button which is clicked as a parameter, throws an exception if the button name is invalid, and returns nothing. The method clickedEndTurn() is called when the Player decides to end their turn. The method neither accepts any parameters nor returns a value. The draggedGameBoard(mousePosition pre, mousePosition post) method is called when the player is panning the game board. This is a void-returning method and it accepts initial and final positions of the mouse. The method clickedMove(int) is called when the Player decides to move their piece to a position. This void-returning method accepts an integer variable which refers to the position at which the piece wants to move. The method clickedShoot(int) is called when the Player decides to shoot at a piece at a position. The method accepts an integer variable which refers to the position at which the piece wants to shoot and does not return any value. All of these methods report the change to the Board Component, and call the methodss accordingly.

The void-returning method clickedExitScreen() is called when the Player wants to quit the match. This method does not accept any parameters and reaches out to both the Display component to change the screen and the Board component indicating that the player wishes to quit the game. The method clickedPanArrow() is called when the Player chooses to switch between the Pan mode and the Arrow mode. This method contacts the Display component to recalculate the board interface on the screen.

2.2.8 Flow Component Class Diagram

The Flow Component Class relationships are summarized in the figure below.

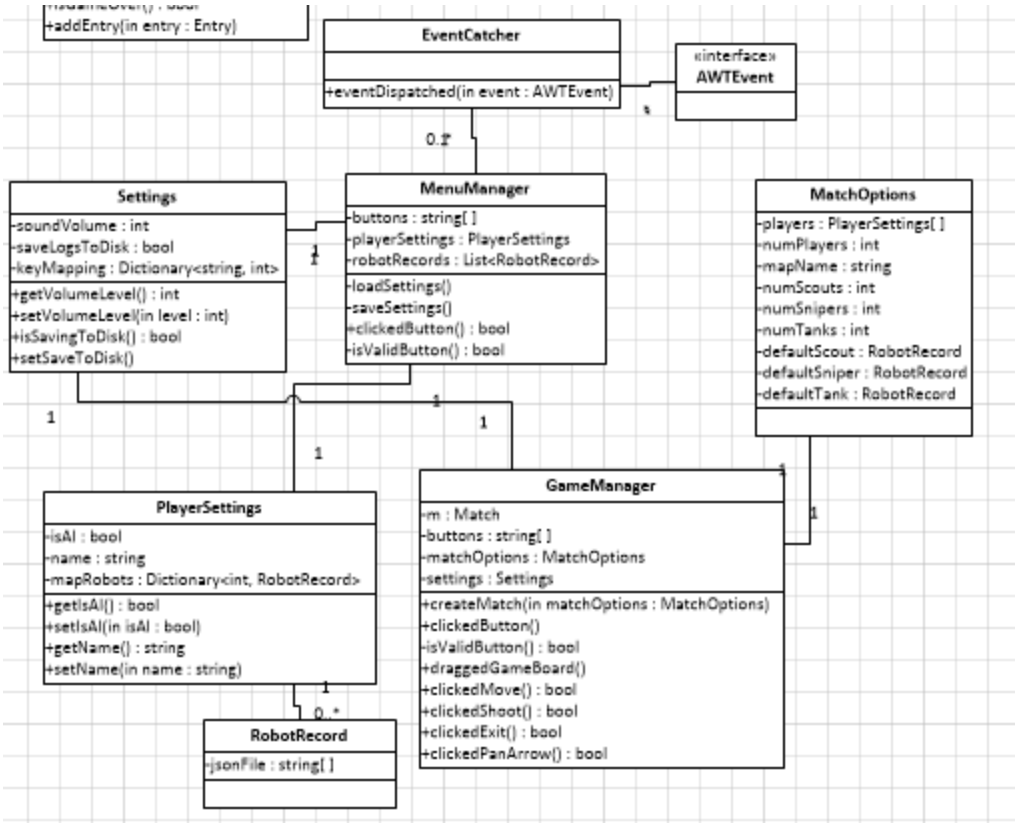

Figure 3: Flow Component Class Diagram

After the Flow Component Classes, the Board Component Classes will be discussed.

2.3 Board Component Classes

The Board Component Classes handle the logical elements of the board state. This includes items like the location of pieces and the results of actions.

2.3.1 Board Class

The Board component is designed to model the game board. The Board class stores an array of the coordinates of the hexagon spaces at which the game pieces are located, an array of the coordinates of the hexagon spaces that are the boundary limits of the game board, an array of the game pieces, the enumeration of the current game piece that is active, the logger manager, the display component, and the interpreter manager. The board component not only stores the necessary coordinates and pieces, but also manipulates the position of each piece according to direct instruction from the event handlers while also checking that each action is valid. The board component notifies the logger manager and the display after each action so that each may update according to how the game board state has changed.

The Board class contains the following fields:

- The HexCoord[] pieceCoords stores a collection of coordinates that represent the locations of the game pieces on the game board. The piece locations are represented by HexCoord elements (see Section 2.3.2), which will store the x-, y-, and z-coordinates of each piece. The array can be indexed using the enumeration assigned to each piece within the current match. This collection will be updated as the game progresses.

- The HexCoord[] boardBounds stores a collection of the boundary limits of the board. The boundary coordinates are represented by HexCoord elements (see Section 2.3.2), which will store the x-, y-, and z-coordinates of each boundary limit. The array may be indexed if the index of the needed boundary is known, but it will mostly be iterated through to check that a hexagon coordinate is within the game board boundary limits.
- The Team[] teams stores the collection of teams. The teams will be represented by the Team class (see Section 2.4). The teams will be manipulated and their state updated as the game is played. The teams will also be used to interact with the interpreters that live in each of the teams. This collection will be manipulated and updated as the game progresses.
- The Piece[] pieces stores the collection of game pieces on the game board. The pieces will be represented by the Piece class (see Section 2.4). The pieces will be manipulated and their state updated as the game is played. This collection will be manipulated and updated as the game progresses.
- The Logger logger stores the logger. The logger will be represented by the Logger class (see Section 2.6). The logger manager will be notified when an action or some critical update that needs to be logged is performed.
- The Display display stores the display. The display will be represented by the Display class (see Section 2.1). The display will be notified when an action or some critical update that needs to be visualized is performed.

The fields in the Board class depend on the definition of the PieceEnum. The PieceEnum is an enumeration, containing a fixed set of values which map to all of the game pieces in the game. It will contain values for each game piece so that each piece has its own unique enumeration. Some values are RED_SCOUT, RED_SNIPER, RED_TANK, BLUE_SCOUT, BLUE_SNIPER, BLUE_TANK, and so forth.

The Board class is responsible for initializing the game, whose state will then be communicated to the display and the logger manager to be displayed and logged, respectively. The Board class will receive instructions from the Flow component (see Section 2.2) in the form of its functions being invoked by the component as events are received and handled.

The Board class has three methods for manipulating the game board. The method movePiece(PieceEnum piece, HexCoord coord) takes in the piece to move and the coordinate to which the piece is to be moved, which must represent a valid coordinate of a hexagon space and be within range of the piece's remaining mobility points. The method shootSpace(HexCoord coord, PieceEnum piece) takes in the target coordinate of the hexagon space being shot, which must be a valid hexagon space and be within range of the piece's range points, and the enumeration of the piece doing the shooting. Finally, the method nextPiece() does not take in any parameters but simply determines the next piece that is to be moved.

Also, the Board class has two method for scanning one space and an area. The method scanArea(PieceEnum piece, HexCoord coord) takes in the enumeration that represents the piece that is performing the scan operation and the target coordinate of the hexagon space, which must be a valid coordinate, that is being scanned and then returns a list of HexCoord elements representing the coordinates that are determined to be occupied. The method scanSpace(HexCoord coord) takes in one coordinate of a hexagon space, which must be valid coordinate, and returns a list of pieces determined to be occupying the space.

Next, the Board class has two methods for converting between absolute and relative coordinates. The method abosuleToRelative(HexCoord coord) takes in one coordinate of a hexagon space, which must be a valid coordinate representing a hexagon space on the game board, and returns a HexCoord element representing the coordinate converted into a relative coordinate. The method relativeToAbsolulte(HexCoord coord) takes in one coordinate of a hexagon space, which must be a valid coordinate representing a hexagon space on the game board, and returns a HexCoord element representing the original coordinate converted into an absolute coordinate.

Lastly, the Board class has a method for checking that a given coordinate is within the boundaries of the game board limits. The method isInBounds(HexCoord coord) takes in one coordinate representing a hexagon space on the game board and returns true if the coordinate is within the boundaries of the game board limits and false otherwise.

2.3.2 HexCoord Class

The HexCoord class is designed to model the hexagon space elements on the game board. The HexCoord class will be used to refer to the actual hexagon spaces on the game board and for storing the coordinate positions of the game pieces. The HexCoord class will also be used throughout the system when it is necessary to refer to a hexagon space on the game board, such as within the display component to update the visual display of a game piece.

The HexCoord class contains the following fields for representing the position of the element:

- The Integer x represents the x-coordinate of the hexagon space. The x-coordinate represents the axis that declines through the hexagon from the left to the right.
- The Integer y represents the y-coordinate of the hexagon space. The y-coordinate represents the axis that rises through the hexagon space from the left to the right.
- The Integer z r
- epresents the z-coordinate of the hexagon space. The z-coordinate represents the axis that runs vertically through the hexagon space.

The HexCoord class's only methods are accessors and mutators for its three fields. It contains getX(), getY(), getZ(), which all return their respective coordinate values for the space element, setX(int x), setY(int y), setZ(int x), which all take the new value for their respective coordinate value of the space element. The HexCoord class also contains the method reduce(), which simply reduces the space element's coordinates to their minimal representation.

2.3.3 Board Component Class Diagram

The Classes for the Board Component are summarized in the figure below.

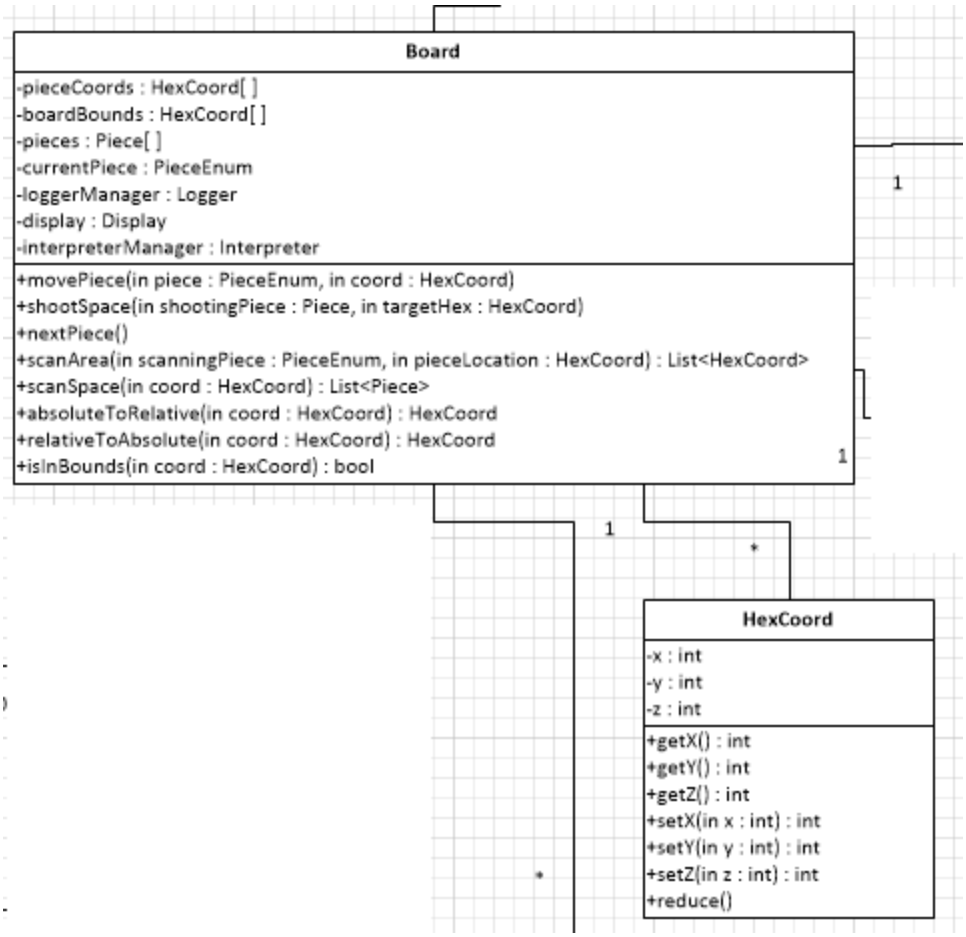

Figure 4: Board Component Class Diagram

2.4 Pieces Component Classes

The Pieces Component Classes are mainly a data store that holds the data relevant to each piece on the board.

2.4.1 Piece Class

The Piece class is mainly a container for holding data regarding the statistics and capabilities of a given piece. It is not responsible for tracking information such as the position of the piece. It serves as mostly an end point of queries.

The Piece class contains the following fields:

- Integer attack: The attack field holds the Piece's attack value, representing how much damage it does when it shoots an enemy.
- Integer range: The range field holds the Piece's range value, representing both how far the Piece can see and how far it can shoot.
- Integer movement: The movement field holds the piece's movement value, representing how many spaces the Piece can move per turn.
- Integer currentMovement: The currentMovement field holds the Piece's current movement value, representing how many more spaces the Piece can move on this turn.

- Integer health: The health field holds the Piece's maximum health value, representing how much total damage the Piece can absorb before it dies.
- Integer currentHealth: The currentHealth field holds the Piece's current health value, representing how much more damage the Piece can absorb before it dies.
- Boolean alive: The alive field represents whether the Piece is currently alive. If it is not alive, it may not move, shoot or otherwise act.
- Integer damageDealt: The damageDealt field is a running total of all of the damage this Piece has dealt to other Pieces (including itself) during this match. It has no impact on the game itself but is stored for updating statistics when the match is over.
- Integer damageTaken: The damageTaken field is a running total of how much damage this Piece has absorbed during this match. It has no impact on the game itself but is stored for updating statistics when the match is over.
- Integer spacesMoved: The spacesMoved field is a running total of how many spaces this Piece has moved during this match. It has no impact on the game itself but is stored for updating statistics when the match is over.
- Integer enemiesDefeated: The enemiesDefeated field is a running total of how many opposing Pieces this Piece has defeated during this match. It has no impact on the game itself but is stored for updating statistics when the match is over.
- Integer turnsTaken: The turnsTaken field is a running total of how many turns this Piece has taken during this match. It has no impact on the game itself but is stored for updating statistics when the match is over.
- Integer absoluteRotation: This field is a representation of the absolute rotation of this piece, given from 0 to 5, on the board.
- bool hasShot: This field determines whether the Piece has shot during its turn. If it has shot once, it may not shoot again.

Most of the methods associated with the Piece class are accessor methods for its field. The Board may query the piece's fields using the methods getAttack(), getRange(), getMovement(), getCurrentMovement(), getHealth(), getCurrentHealth(), getDamageDealt(), getDamageTaken(), getSpacesMoved(), getEnemiesDefeated(), getTurnsTaken(), getAbsoluteRotation, getHasShot() and isAlive().

Some of the Piece's fields are modified during its turn. The reset() method allows these values (hasShot and currentMovement) to be reset, so that the Piece's next turn may be taken correctly.

During a turn, the Board component handles the spatial logic of a Piece moving and shooting, but the Piece must still handle these actions internally in order to update its statistics correctly. The shoot() method takes in Integers representing the amount of damage dealt and enemies defeated on a shot, and updates these statistics, also setting hasShot to true for this turn. The move() method takes two integers, representing the number of spaces moved and the relative direction of movement, and it decrements the value of currentMovement, updating statistics as appropriate. The takeDamage() method is called when the Piece is shot, and takes in an integer representing the amount of damage received. It calculates the Piece's remaining health and returns the value of the alive field. All of these

methods will also send a message to the GameScreen class (see Section 2.1.3), informing the Display that the Piece has performed some action in order to animate it properly.

The remaining methods in the Piece class concern the rotation of the Piece, which has no impact on the logic of the Board. The getRelativeRotation() method takes an integer representing some absolute rotation value. It compares the value given with this Piece's absoluteRotation value, and returns the relative rotation from the Piece's rotation to the passed-in rotation, which will be a value between 0 and 5 (that is, how many times the Piece must turn to reach the target rotatation). Conversely, the getAbsoluteRotation() method takes an integer representing some relative rotation value between -5 and 5. It checks this value against the Piece's absoluteRotation, and returns the absolute rotation of the passed-in value. The rotate() method actually updates the absoluteRotation value of the piece, taking in a relative rotation value between -5 and 5, calling getAbsoluteRotation(), and setting the piece's absoluteRotation to the return value. It will also send a message to the GameScreen class, informing the display that the Piece has updated its rotation.

2.4.2 Team Class

The Pieces are all stored in a Team class, which represents a group of Pieces working in tandem to win the match. It holds all the Pieces which are working together, the colour of the Team that is being represented, and the Interpreter associated with the Team, as well as values that help it take its turns properly.

The Team class contains the following fields:

- Piece[] pieces: The pieces field is an array of Pieces, indexed by the PieceEnum. It contains the Pieces for this team.
- Boolean eliminated: This field is a boolean value representing whether this Team has been eliminated or not. A Team is eliminated when all of its Pieces are dead.
- TeamEnum colour: The colour field is a TeamEnum value representing the colour of the team, as described below.
- Interpreter interpreter: The interpreter field holds the Interpreter for this Team (see Section 2.5.1). If this Team is controlled by a human player, the value of this field will be NULL.
- Integer activePiece: The activePiece field is an integer representation of which piece is currently active and taking a turn. It can be used to index the pieces array. This value is set to -1 if there is no active piece.

Some values in the Team class depend on the PieceEnum. The PieceEnum is an enumeration containing a fixed set of values which map to the three types of piece in the game. It will contain the values SCOUT, SNIPER and TANK.

The Team's colour is given by the value of the TeamEnum. The TeamEnum is an enumeration containing a fixed set of values which map to the six colours of teams in the game, in the order that they may take their turns. It will contain the values RED, ORANGE, YELLOW, GREEN, BLUE, and PURPLE.

The Team class contains a method for getting the next active piece. getNextPiece() will examine each Piece in the pieces array after the last value of activePiece, and calls setActivePiece() to assign the value of activePiece to that index if it is able to take a turn. If no Piece is able to take a turn, it calls the

setEliminated() method, which sets its eliminated field to true. The isEliminated() method will return the value of the eliminated field.

The Team class can return its own Interpreter with the getInterpreter() method, and can order the Interpreter to run a piece's play command with the playAI() method, which takes an integer index as a parameter and runs that Interpreter's play() method on the correct piece.

2.4.3 Pieces Component Class Diagram

Below can be found the Class Diagram for the Classes in the Pieces Component.

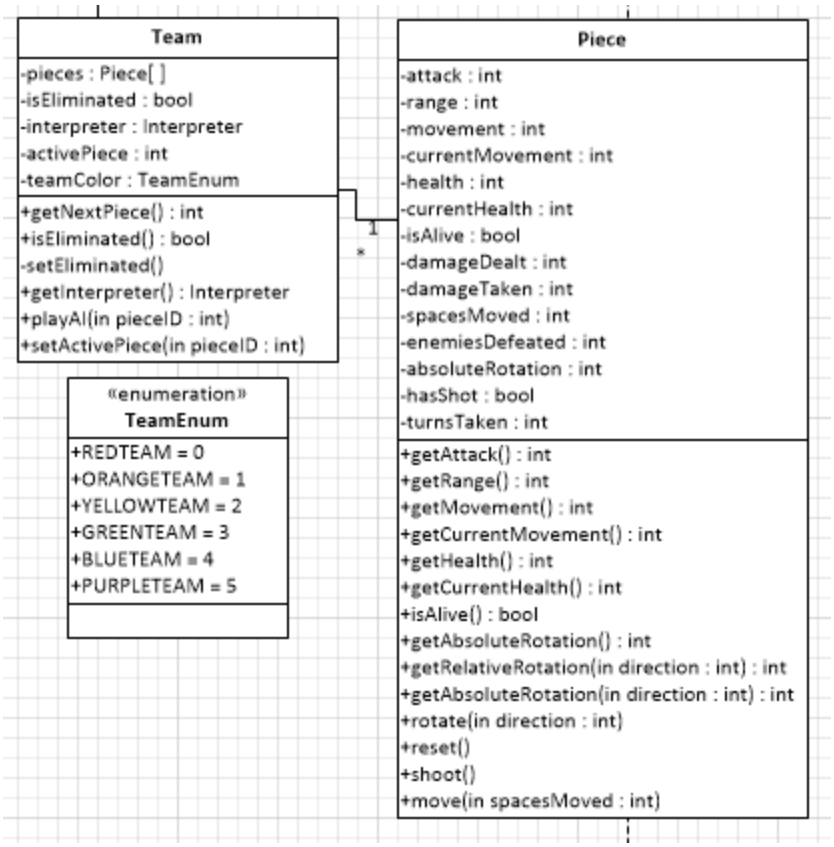

Figure 5: Pieces Component Class Diagram

2.5 Interpreter Component Classes

The Interpreter Component Classes handle parsing and executing the AI instruction files obtained from the Robot Librarian.

2.5.1 Interpreter Class

The heart of the Interpreter component is the Interpreter class. Each computer-controlled Team is assigned an Interpreter, which holds the AI programs for each piece on the team and the Mailboxes for each piece, as well as the data structures that each program will need to use to store its data when it runs. The Interpreter also contains a large number of library functions for the programs as specified in the RoboSport370 language specification document, including mathematical operators and

comparisons, and functions for querying the board. When the Interpreter runs an AI program, it uses an integer index to access the relevant data structure for the current piece.

The Interpreter class contains the following fields:

- List<UserWord>[] userWords: This field is an array of Lists of UserWords (see Section 2.5.5), indexed by the piece. Each piece's program will define a number of UserWords, which are used as shorthand for several values or commands.
- List<UserVariable>[] userVars: This field is an array of Lists of UserVariables (see Section 2.5.4), indexed by the piece. Each piece's program will define a number of UserVariables, which may store values for later.
- Mailbox[] mailboxes: The mailboxes field is an array of Mailboxes (see Section 2.5.3), indexed by the piece. Each program will be able to send and receive messages from other pieces, and each mailbox stores these messages.
- AI[] ai: The ai field is an array of AIs (see Section 2.5.2), indexed by the piece. The AI is where the raw and cleaned program data is stored, and this data is copied from the relevant AI whenever that piece's program needs to be run.
- Stack[] stacks: This field is an array of Stacks, as defined in the Java standard library, which is indexed by the piece. The Forth-like language of the program requires values to be pushed to and popped from a stack data structure. Each program will need to maintain a stack of values as it executes.
- Integer currentPiece: The currentPiece represents the piece which is currently executing its program. This value is used as the index for the arrays of values above.
- Integer numberOfPieces: The numberOfPieces represents the total number of AI pieces on this team. The default value is 3.
- Dictionary<String, Integer> idToIndex: The AI programs need to refer to pieces by a string identifier, such as "SNIPER." This field contains a mapping of these string IDs to integer indexes that can be used to access the arrays defined above.
- List<Integer> loopIterators: Some programs which loop require a variable to keep track of the current iteration, and these loops may be nested. This field holds the current iterator value for each level of a loop, and is cleared at the end of each command's execution.
- List<String> currentInstructions: This field holds a list of strings which need to be parsed to execute some command. These strings are typically retrieved from an AI (see section 2.5.2)
- Dictionary<String, Integer> functionTableMapping: This field holds a dictionary of strings, where each string is a basic word defined in the Interpreter library (such as "multiply"), and the corresponding integer is the index of the function in the function table.
- Word[] methods: This field holds an array of Words (defined below), which will hold references to the function to be called, effectively creating a function table for the standard Interpreter library of functions, such as mathematical operators, which operate on a Stack.

2.5.1.1 Word Interface

The Word interface is a simple Java interface containing only one undefined method, call(). When the Interpreter is initialized, it creates an array of Words and defines the call() method for each of them as a method matching some standard Interpreter method, effectively creating a function table.

The Interpreter class defines 38 methods which have their entries in the methods function table, all operating on the a Stack (the stacks field indexed by the currentPiece field):

- add() : The add() method will pop two values from the stack, add them together, and push the result onto the stack. It will push nothing if there are not enough values on the stack, or if the two values are not integers.
- subtract() : The subtract () method will pop two values from the stack, subtract the second from the first, and push the result onto the stack. It will push nothing if there are not enough values on the stack, or if the two values are not integers.
- multiply() : The multiply() method will pop two values from the stack, multiply them together, and push the result onto the stack. It will push nothing if there are not enough values on the stack, or if the two values are not integers.
- divideRemain() : The divideRemain() method will pop two values from the stack, divide the first value into the second, and then push the remainder and quotient, in that order. It will push nothing if there are not enough values on the stack, or if the two values are not integers.
- and() : The and() method will pop two values from the stack and compare them. If both are true, it will push “true,” otherwise it will push “false.” It will push nothing if there are not enough values on the stack, or if the two values are not boolean.
- or() : The or() method will pop two values from the stack and compare them. If either one is true, it will push “true,” otherwise it will push “false.” It will push nothing if there are not enough values on the stack, or if the two values are not boolean.
- invert() : The invert() method will pop one value from the stack. If it is a boolean, it will invert it (eg. “true” becomes “false”), and then it will push the result on the stack. It will push nothing if there are not enough values on the stack, or if the value is not boolean.
- duplicate() : The duplicate() method will pop one value from the stack, and then push that value onto the stack two times. It will do nothing if there are no values on the stack.
- drop() : The drop() method will pop one value from the stack and discard it, pushing nothing.
- swap() : The swap() method will pop two values from the stack, and then will push the first value onto the stack, followed by the second, in order to swap their order. It will push nothing if there are no values on the stack, and one value if there is only one value on the stack.
- rotate() : The rotate() method will pop three values from the stack. It will then push the second value, followed by the first value, followed by the third value. It will push the first and second values back onto the stack in their original order if there are not enough values on the stack.
- greaterThan() : The greaterThan() method will pop two values from the stack and compare them, pushing the result onto the stack. It will push nothing if there are not enough values on the stack or if the two values cannot be compared.
- greaterThanEqual() : The greaterThanEqual() method will pop two values from the stack and compare them, pushing the result onto the stack. It will push nothing if there are not enough values on the stack or if the two values cannot be compared.

- lessThan() : The lessThan() method will pop two values from the stack and compare them, pushing the result onto the stack. It will push nothing if there are not enough values on the stack or if the two values cannot be compared.
- lessThanEqual() : The lessThanEqual() method will pop two values from the stack and compare them, pushing the result onto the stack. It will push nothing if there are not enough values on the stack or if the two values cannot be compared.
- equal() : The equal() method will pop two values from the stack and compare them, pushing the result onto the stack. It will push nothing if there are not enough values on the stack or if the two values cannot be compared.
- notEqual() : The notEqual() method will pop two values from the stack and compare them, pushing the result onto the stack. It will push nothing if there are not enough values on the stack or if the two values cannot be compared.
- if() : The if() method will be called if the parser detects a conditional block. It will execute comparison methods listed above for the test condition, and then execute some instructions based on the result of the test condition, handling the control flow.
- while() : The while() method will be called if the parser detects a guarded loop block. It will execute comparison methods listed above for the test condition, and then execute some instructions based on the result of the test condition, handling the control flow. It will append a new loop iterator onto the loopIterators list in the Interpreter when it begins, increment it as it loops, and remove that iterator when it ends.
- for() : The for() method will be called if the parser detects a counted loop block. It will execute comparison methods listed above for the test condition, and then execute some instructions based on the result of the test condition, handling the control flow. It will append a new loop iterator onto the loopIterators list in the Interpreter when it begins, increment it as it loops, and remove that iterator when it ends.
- declareVar() : The declareVar() method will be called if the parser detects the definition of a new user-defined variable. It will take the information in the instructions and create a new UserVariable instance, adding it to the appropriate list.
- declareWord() : The declareWord() method will be called if the parser detects the definition of a new user-defined word. It will take the information in the instructions and create a new UserWord instance, adding it to the appropriate list.
- random() : The random() method will pop an integer from the stack, and push a random integer value to the stack that lies between 0 and the popped value. It will push the original value back if it is not an integer, and do nothing if there are no values on the stack.
- dotPrint() : The dotPrint() method will pop the value at the top of the stack and print it to the console.
- qHealth() : The qHealth() method is used by a program to query the current piece's health. The Interpreter will query the Board and push the value onto the stack.
- qHealthLeft() : The qHealth:eft() method is used by a program to query the current piece's remaining health. The Interpreter will query the Board and push the value onto the stack.
- qMoves() : The qMoves() method is used by a program to query the current piece's movement. The Interpreter will query the Board and push the value onto the stack.
- qMovesLeft() : The qMovesLeft() method is used by a program to query the current piece's remaining movement. The Interpreter will query the Board and push the value onto the stack.

- qAttack() : The qAttack() method is used by a program to query the current piece's attack. The Interpreter will query the Board and push the value onto the stack.
- qRange() : The qRange() method is used by a program to query the current piece's range. The Interpreter will query the Board and push the value onto the stack.
- qTeam() : The qTeam() method is used by a program to query the current piece's team. The Interpreter will query the Board and push the value onto the stack as a string.
- qType() : The qType() method is used by a program to query the current piece's type (SCOUT, SNIPER or TANK). The Interpreter will query the board and push the value onto the stack as a string.
- turn() : The turn() method pops an integer value off the stack, and the Interpreter directs the Board to rotate the piece the relative value. It does nothing if the value is not an integer.
- move() : The move() method has the Interpreter direct the Board to move the piece one square in the direction it is facing.
- shoot() : The shoot() method pops two integer values off the stack representing range and relative direction. It then has the Interpreter direct the Board to handle the piece's shot in the given direction and range. If the piece has already shot, nothing happens. The method does nothing if there are not enough values on the stack or if the values are not integers.
- check() : The check() method pops an integer value off the stack, and the Interpreter queries the Board to ask what is contained in the space in that given direction. The result will be a string (EMPTY, OUT OF BOUNDS or OCCUPIED) that is pushed to the stack. The method does nothing if there are no values on the stack or the value is not an integer.
- scan() : The scan() method has the Interpreter query the Board about which pieces, if any, are in visible range. The scan order must be identical each time. It will push an integer value onto the stack representing the number of visible robots.
- identify() : The identify() method pops an integer value from the stack, and has the Interpreter query the Board about the nth piece detected when scanning the Board. Four values will be pushed to the stack: the piece's remaining health, its direction relative to this one, its range, and its team colour. This method does nothing if there are not enough values on the stack or if the value is not an integer.

The above methods can be considered the "standard library" of the Forth-like language.

The Interpreter class has three methods which are not in the library of functions for programs: setCurrentPiece(), play(), and parse(). setCurrentPiece() takes an integer and sets the value of currentPiece to that integer, controlling which piece is active. The play() method loads the play program of the currentPiece into the currentInstructions list, and then calls the parse() method. The parse() method systematically examines each term in the currentInstructions list, parsing it into a value or command, and then removes it from the list. It continues until the list is empty, at which point the command has finished executing.

2.5.2 AI Class

For the Interpreter to work, it must have data to operate on. The AI class holds the programs that will be executed by the Interpreter for each piece. The AI takes a reference to a RobotRecord in its constructor.

When the AI is initialized, this program is split into two sections: initialization and play. These are cleaned and converted into a list of strings representing commands and values.

The AI class contains the following fields:

- String[] fullProgram: The fullProgram contains the raw text of the program, as it was received from the RobotRecord in its constructor.
- List<String> splitPlayProgram: The splitPlayProgram field contains the lines of the fullProgam field that pertain to the piece's play command, with the comments removed and each term separated into its own entry in the list. This list can be copied whenever the piece needs to execute its play command, and the interpreter will go through the terms one by one.
- List<String> splitInitProgram: The splitInitProgram field contains the lines of the fullProgam field that pertain to the piece's initialization, with the comments removed and each term separated into its own entry in the list. These lines of code are executed to create the user-defined words and variables that the program might need when it executes its play command. The interpreter retrieves this list and parses it before the game begins.

The AI class populates all of its fields in its constructor, and has no mutator methods. It has methods for accessing the data in each field: getFull() will return the raw version of the AI's program, while getInit() and getPlay() will return the cleaned sections of the program. It also contains the stripComments() method, which takes a string and returns that string with the Forth comments removed, as a helper function to assist in cleaning the raw program data.

2.5.3 Mailbox Class

The different AI programs need a way to communicate, and the Mailbox class is designed to allow message passing between programs. Each mailbox is assigned to one piece, and will contain string messages received from pieces. A piece may send messages to itself. Each mailbox has a hard limit on the number of messages that may be stored, and if a mailbox is full it may not receive any more messages.

The Mailbox class contains the following fields:

- List<String> messages: The messages field holds the messages received from other pieces. Each message is a string, which contains the ID of the sender, a separation character, and then the message itself.
- Integer mailboxSize: mailboxSize defines the limit on the number of messages that may be held in the mailbox at once.
- String pieceID: The pieceID field holds the ID of the piece that the Mailbox belongs to.
- Interpreter interpreter: This field holds a reference to the interpreter holding the Mailbox.

The Mailbox class contains accessor and mutator methods only for its pieceID field: getID() will return the ID and setID() will take a string and update the value of pieceID. All of its other methods are associated with handling messages.

The hasMessage() method takes in a string representing a pieceID and searches the messages field to see if there is a message from that piece currently held in the Mailbox. It returns true if one or more

messages are found, and false otherwise. hasMessage() does not validate the pieceID string, and will simply return false if it finds no matches.

If a message exists from a sender, the receiveMessage() method will take a string representing the sender's pieceID, and return a string containing the earliest held message from that sender, or an empty string if the sender's message is not found. If a message is found and returned, receiveMessage() will remove the message from the messages field, because the message has been successfully delivered.

The Mailbox class can also be used to send messages to other Mailboxes. sendMessage() takes as a parameter a string holding the ID of the Mailbox to send to, and a string representing the message to be delivered, appends the pieceID associated with this mailbox, and then uses the reference to the parent Interpreter to deliver the message to the recipient. A Mailbox can send messages to itself.

For convenience, the Mailbox class also contains methods for checking if the messages list is empty, and for emptying the Mailbox without handling any of the messages. The isEmpty() method returns a boolean value of true if the messages list contains any messages, and false otherwise. The clear() method removes all strings from the messages list without performing any operations on them.

2.5.4 UserVariable Class

The Interpreter component contains two classes that are used as containers to hold data that assists in the parsing of user-defined elements of programs. First, the UserVariable class is a container class intended to allow the interpreter easy access to user-defined variables within the robot programs. It only contains its name and value, and methods for accessing them.

The UserVariable class contains the following fields:

- String varName: This field contains the name of the user-defined variable.
- String varValue: This field contains the value of the user-defined variable. It may hold an integer, Boolean value or string, but all values are stored as strings and parsed by the interpreter when necessary.

The UserVariable contains the methods getName() and getValue() for accessing its name and value. It also a has a method setValue() which allows the stored value to be updated. Once the UserVariable has been created, its name may not be changed.

2.5.5 UserWord Class

The other container class used to help parse user programs is the UserWord class. UserWord is a container class intended to allow the interpreter an easy way to store user-defined words. These words represent a series of values or words which are added to the command stream to replace the given word.

The UserWord class contains the following fields:

- String wordName: This field contains the name of the user-defined word, as it is given in the user program.
- List<String> replaceValues: This field contains a list of the strings, which would be split on whitespace, that will replace the given word in a program line as it is parsed. These are inserted

in the Interpreter's currentInstructions field to replace the word whenever it is encountered by the parser.

UserWord contains the methods getName() and getReplaceValues() for accessing its name and replace values. It also a has a method setReplaceValues() which allows the stored replacement values to be updated. Just like the UserVariable class, once the UserWord has been created, its name may not be changed.

2.5.6 Interpeter Component Class Diagram

Below is the Class Diagram for the Interpreter Component Classes.

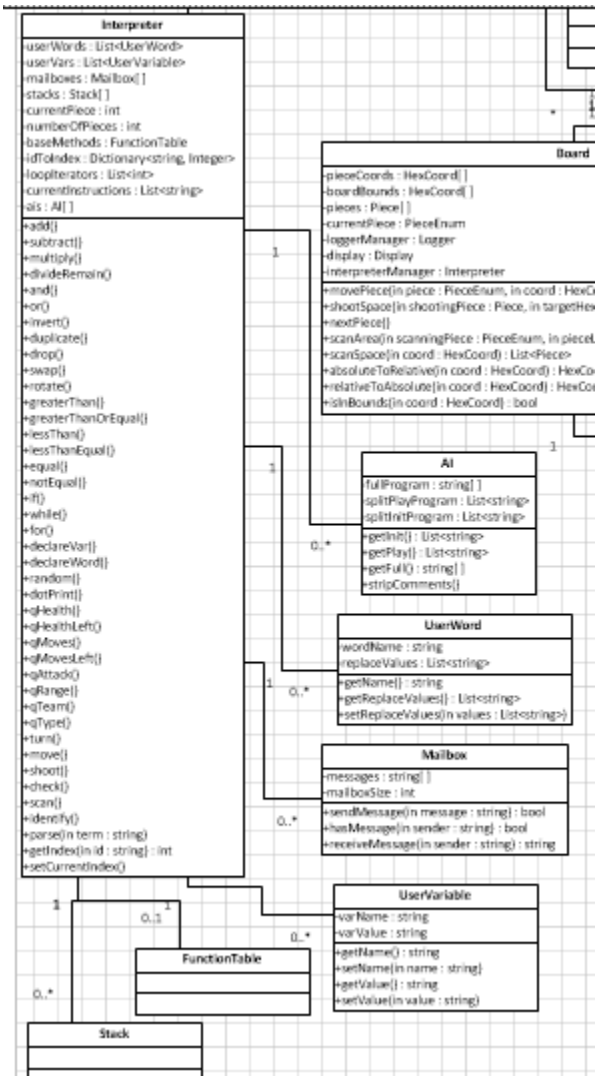

Figure 6: Interpreter Component Class Diagrams

2.6 Logger Component Classes

The final Classes detailed are the Classes related to the Logger component. These Classes both hold textual information about the game progress but also several tags that allow for filtering the results.

2.6.1 Logger Class

The Logger class is designed to model the logger component. The class stores an array of seven logs (one for all the logs and six for each team in six-player game), or four logs (one for all the logs and three for each team in three-player game), or three logs (one for all the logs and two for each team in two-player game). This can be easily indexed using the TeamEnum enumeration that is assigned to each of the teams of the Team class.

The Logger class contains one private field:

- Log[] players: The players field is an array for Log elements (see Section 2.6.2). It contains all of the logs in the game which will be displayed, indexed by the value of the TeamEnum (defined below).

The field in the Logger class depends on the definition of the TeamEnum. The TeamEnum is an enumeration, containing a fixed set of values which map to all of the logs in the game. The values in the TeamEnum are from PlayerSelection screen, such as RED, PURPLE, BLUE, GREEN, YELLOW and BROWN.

The Logger class is responsible for displaying all of the logs in the game, which will be displayed to the user. It has five different types of display options, so it contains the following methods for each type of display:

The displayLog() method takes a TeamEnum value representing the current team, and is for displaying all of the logs for this team, includes movement, shooting and damaged/death/game over.

The displayMovement() method takes a TeamEnum value representing the current team, and is for displaying all of the movement logs for this team. For example, Scout moves from (a, b, c) to (x, y, z).

The displayShooting() method takes a TeamEnum value representing the current team, and is for displaying all of the shooting logs for this team. For example, Scout shoots (x, y, z), kills Blue Team Scout and damages Green Team Tank (Health: 3 -> 2)!

The displayDamaged() method takes a TeamEnum value representing the current team, and is for displaying all of the damaged logs for this team. For example, Your Tank gets 1 point damage from Red Team or Your Scout is killed by Red Team.

The clear() method is for clearing all the current logs for the user.

2.6.2 Log Class

The Log class is designed to model logs for one team or for all teams. Each team will have different logs based on the viewing scope. The log for all teams is for letting AI make smart decisions.

The Log class contains the following fields:

- List<Entry> teamLog: The teamLog field is a list of Entry elements (see Section 2.6.3). It contains all of the entries in the game.
- List<Integer> teamMovement: The teamMovement field is a list of integers, which stores the numbers that indicate the movement.
- List<Integer> teamShooting: The teamShooting field is a list of integers, which stores the numbers that indicate the shooting.

- List<Integer> teamDamaged: The teamDamaged field is a list of integers, which stores the numbers that indicate the damaged and death.
- Boolean[] isPieceAlive: The isPieceAlive field is an array of three Boolean values representing Scout, Snipper and Tank, which stores whether each piece is alive or not. When all of the piece are dead, the game of this user is over.

The Log class is responsible for providing Logger class with the logs of each team in the game. It contains the following methods:

The getTeamLog() method returns all logs of this team in the string format.

The getTeamMovement() method returns all the movement logs of this team in the string format based on the private teamMovement field.

The getTeamShooting() method returns all the shooting logs of this team in the string format based on the private teamShooting field.

The getTeamDamaged() method returns all the damaged or death logs of this team in the string format based on the private teamDamaged field. It also calls the isGameOver() method every time to determine whether the game of this user is over or not. When isGameOver() returns true, it adds *Game Over!* to the returning string.

The isGameOver() method returns whether the game of this user is over by determining whether all of the pieces are dead based on the private isPieceAlive field.

2.6.3 Entry Class

The Entry class is designed to handle one entry, and it gets information from the board component. The class stores one entry, and uses EntryEnum enumeration to indicate different types of entries.

The Entry class contains the following fields:

- String entry: The entry field stores one entry.
- EntryEnum currentEntry: The currentEntry stores an enumeration value representing the current type of entry. The EntryEnum is an enumeration, containing a fixed set of value which map to all types of the entries in the game. It contains MOVEMENT, SHOOTING and DAMAGED.

The Entry class is responsible for handling different types of entries, so it contains the following methods:

The addEntry() method handles different types of entries and prepares to add them to corresponding team(s). It stores the string to the entry field from the setMovement() method, the setShooting() method and the setDamaged() method.

The setMovement() method takes two HexCoord values of start and end location and a TeamEnum and a PieceEnum representing the piece from one team moving from one location to another location.

The setShooting() method takes a HexCoord of the target location, a TeamEnum and a PieceEnum representing the piece from one team shooting a target location.

The setDamaged() method takes a HexCoord, a TeamEnum, a PieceEnum and an integer representing all the pieces in a location getting damage of certain value.

The getMovement() method, the getShooting() method and the getDamaged() methods take indexes to return one corresponding entry of a piece in a team.

2.6.4 Logger Component Class Diagram

The Logger Component Class Diagram can be found below.

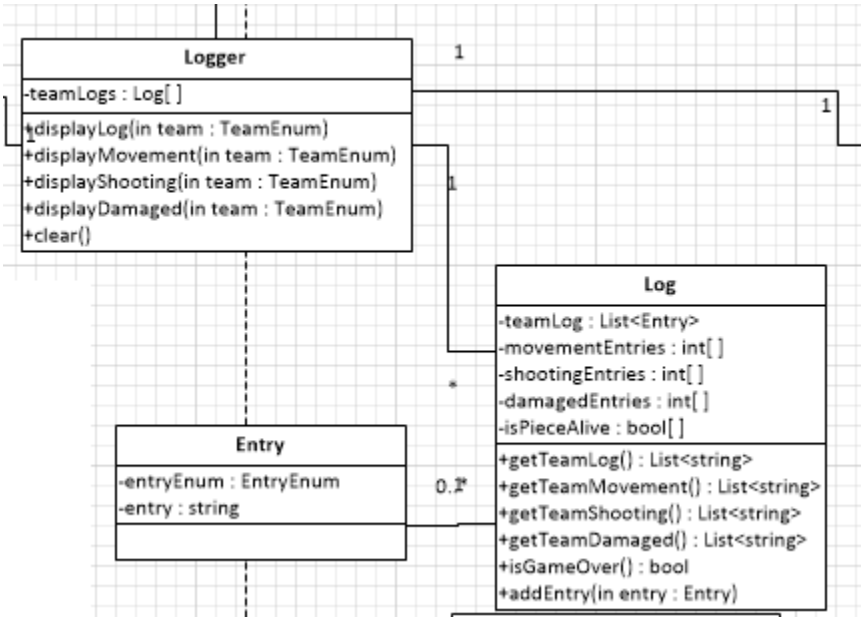

Figure 7: Logger Component Class Diagram

2.7 The Complete Class Structure

Putting all of the various Class diagrams together, a complete Class Structure for the Robo-Wars game is attained. A view of it can be seen below.

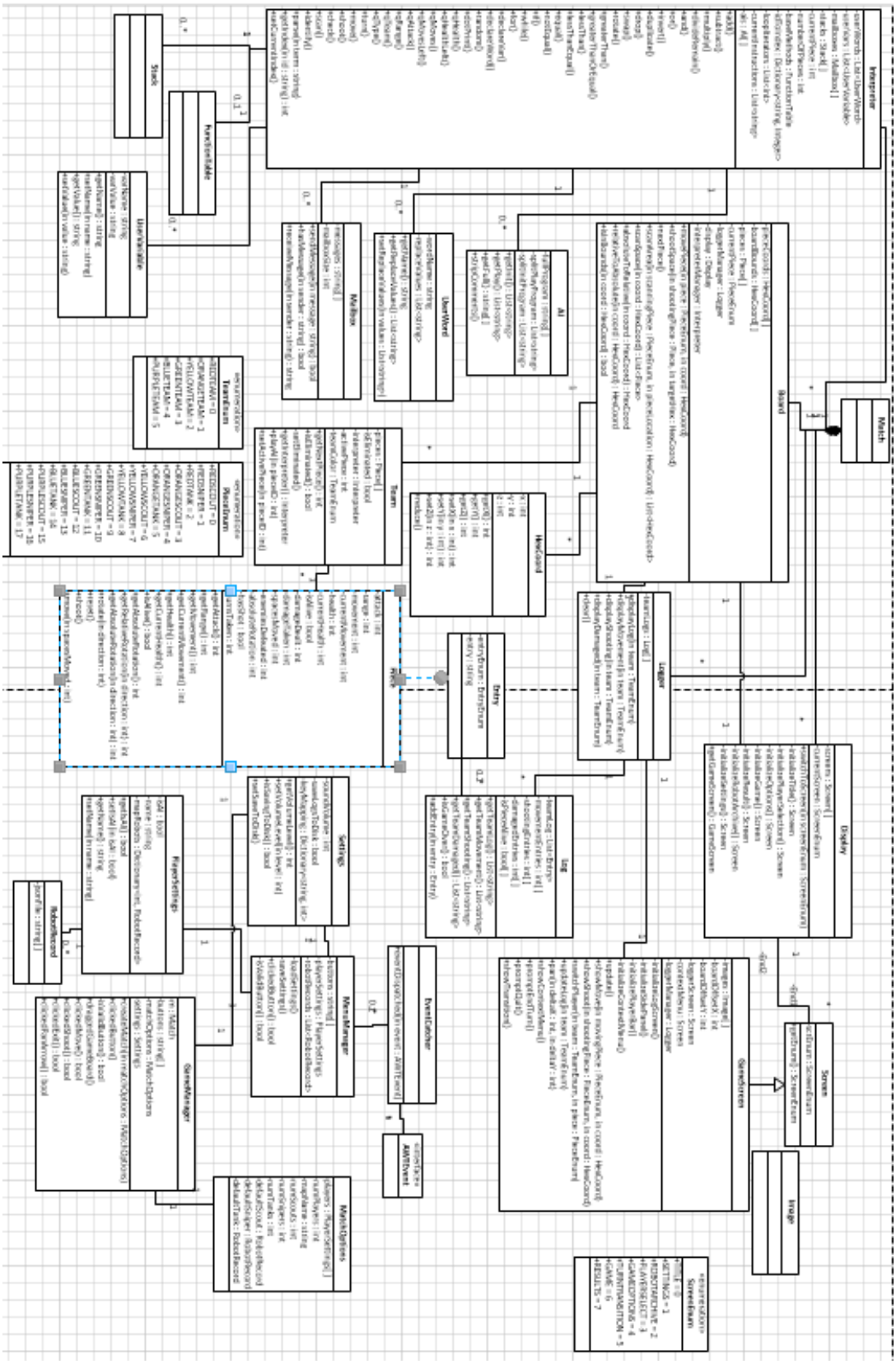

Figure 8: Complete Class Diagram

3.0 Changes to Requirements Document

Upon further inspection of the requirements, the Requirements Document has undergone the following changes.

3.1 Reduction in the Robot Librarian Actor Scenarios

The Robot Librarian Actor is a separate system that interacts with the game. The Scenarios listed in sections 4.6 are actually internal interactions of the Robot Librarian. Therefore, sections 4.6.1-4.6.6, with the exception of 4.6.2 (Download) are removed from the Requirements Document.

Further, 4.6.2 is renamed 4.6.1, and the flow changes as follows. When the game needs a Robot, it will send a request to the Robot Librarian who then fills the request. This is the primary use-case and flow involving the Robot Librarian. There is also the reverse operation, Upload. In this use-case, the game will send to the Robot Librarian files to update the Robot statistics.

In both cases, the format of the file exchange will be JSON files.

3.2 Zooming Moved to Option

Another change to the Requirements Document is the removal of Section 4.3.2.2 from the document. This is the Zoom Board scenario. Currently, it is better considered as an option, as this is technically much more difficult than a Pan scenario. It is now Section 7.2.2 as a Scenario of Additional Game Mechanics in the Options section of the document.

Index