Robo-Wars

Programmer's Manual

Contents

Document Purpose

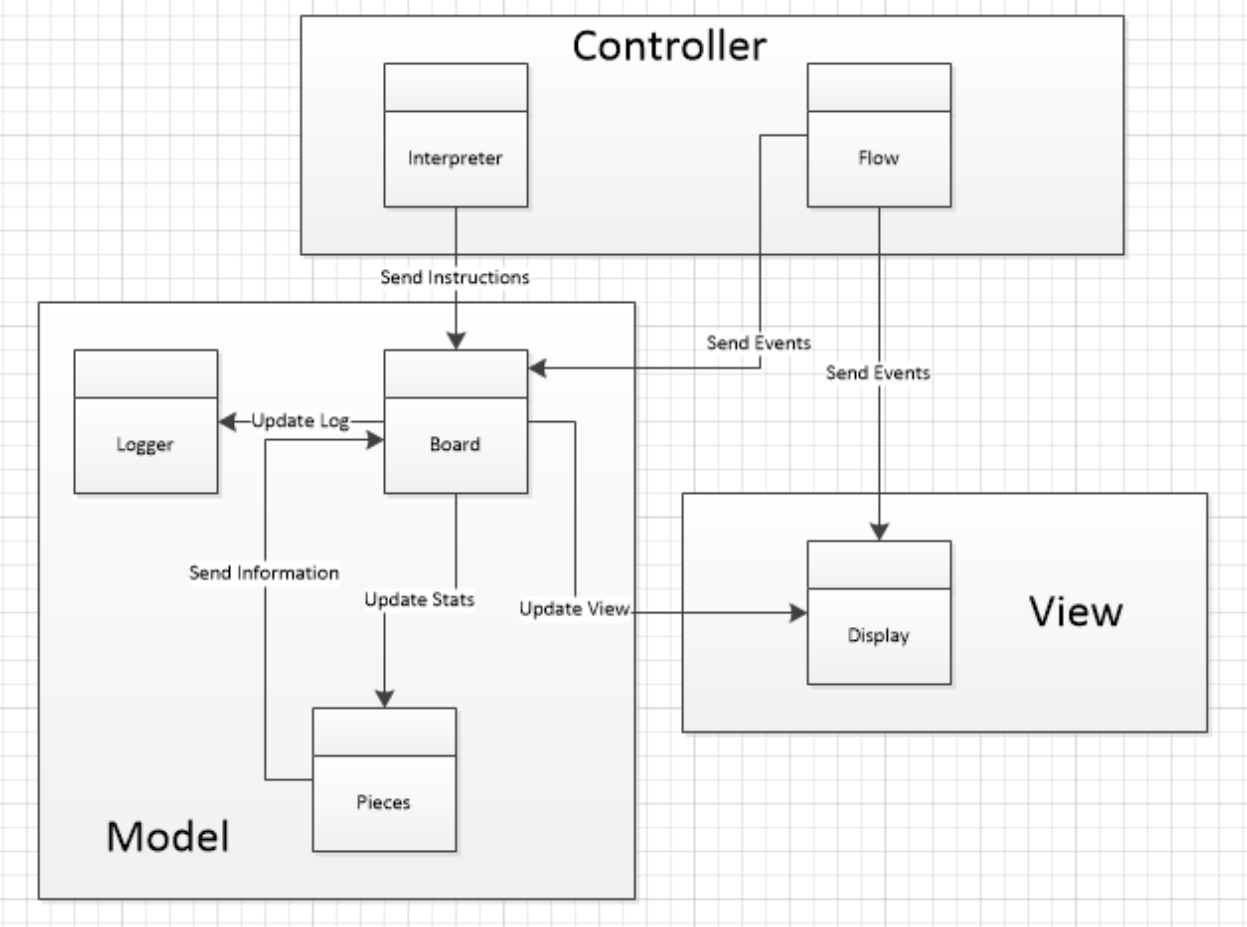
The purpose of this document is to orient a new developer working on the Robo-Wars project on critical or otherwise tricky pieces of code so that the developer can be brought up to speed very quickly. Specific code pieces of each component of the game system have been documented here by the developer that originally constructed the code. Also note that an as-built UML diagram can be found accompanying this document for further reference. Having said that, the first component to be discussed is the Board component.

Architecture Overview

The Robo-Wars project was constructed using an Independent Components architecture, which changed very little from design to completion. It contains six major components:

- The Board component contains the game logic and the primary representation of the game state.
- The Pieces component contains information about each piece in a match, and is primarily used for holding data.
- The Interpreter component is responsible for parsing AI programs for computer-controlled players.
- The Logger component tracks every action taken in a match, for record-keeping purposes or to simulate a completed match over again. This component was not properly integrated into the final system.
- The Flow component handles user-generated events, such as mouse clicks and keyboard presses.
- The Display component is responsible for drawing the application screens and creating a correct visual representation of the board state.

The Flow and Display component closely correspond to the Controller and the View in a Model-View-Controller architecture. Though each component was designed separately in order to reduce dependencies, the system can be viewed through a pseudo-MVC lens, as diagrammed below.



The Board Component

Before working with the Board, it is important to understand how the coordinate system works, as well as how distance is calculated.

The Coordinate System

The hexagons in the board are positioned using a coordinate system. Because the individual position is a hexagon, there are three axes (shown in Figure 1), x, y and z.

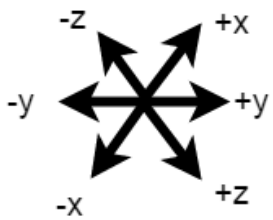


Figure 1: Coordinate Axes

The origin of the axis is in the middle of the board. The hexagon in the middle of the board is (0, 0, 0) as shown in Figure 2 below.

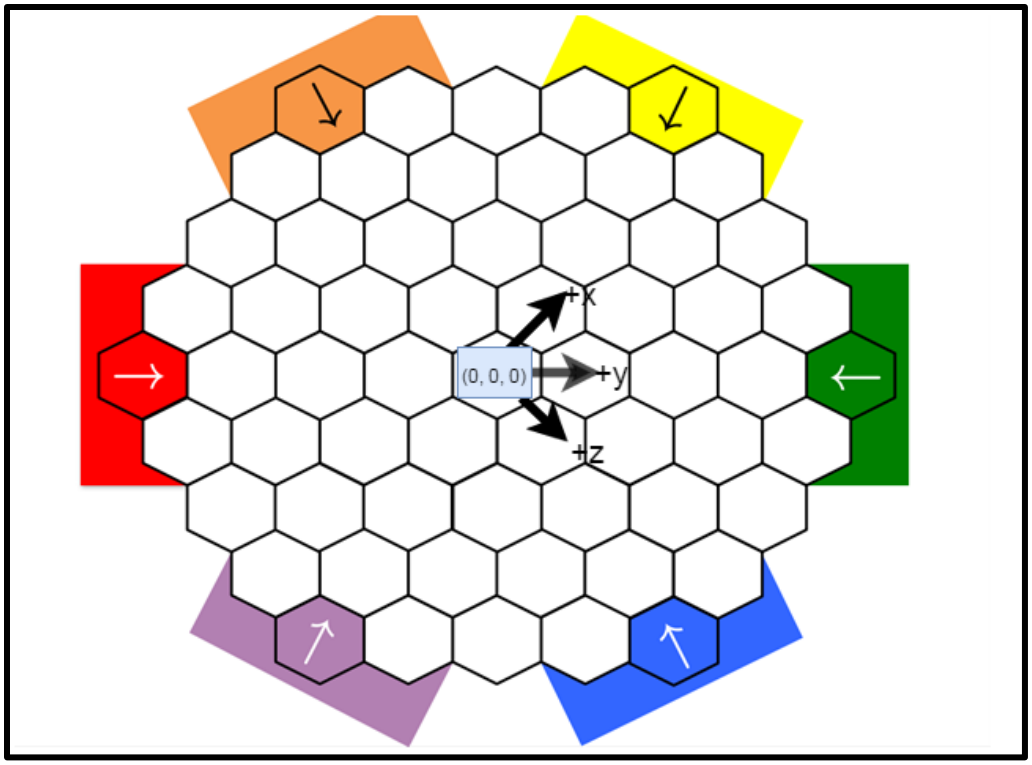


Figure 2: Board with Axes Shown

Therefore, the coordinate for each hexagon is relative to the origin at the centre of the board. The format is (x-coordinate, y-coordinate, z-coordinate).

Notice that 1 unit in the x-direction and 1 unit in the z-direction equals 1 unit in the y-direction.

The system exploits this idea to have a consistent and unique coordinate for every hexagon in the board. Every hexagon in the board can solely be written in the form of x and z coordinate, without a y coordinate. A form of the coordinate where the y-coordinate is 0 is called reduced form, and there exists a method in the HexCoord class, reduce(), which converts a coordinate into its reduced form.

It is very tempting to throw away the y-coordinate and work with just the x and z coordinates. However, the y-coordinate is needed for other purposes such as calculating the shortest distance, which is why we are allowing the y-coordinate to exist.

The following figure contains the unique coordinates in the reduced form given to each hexagon.

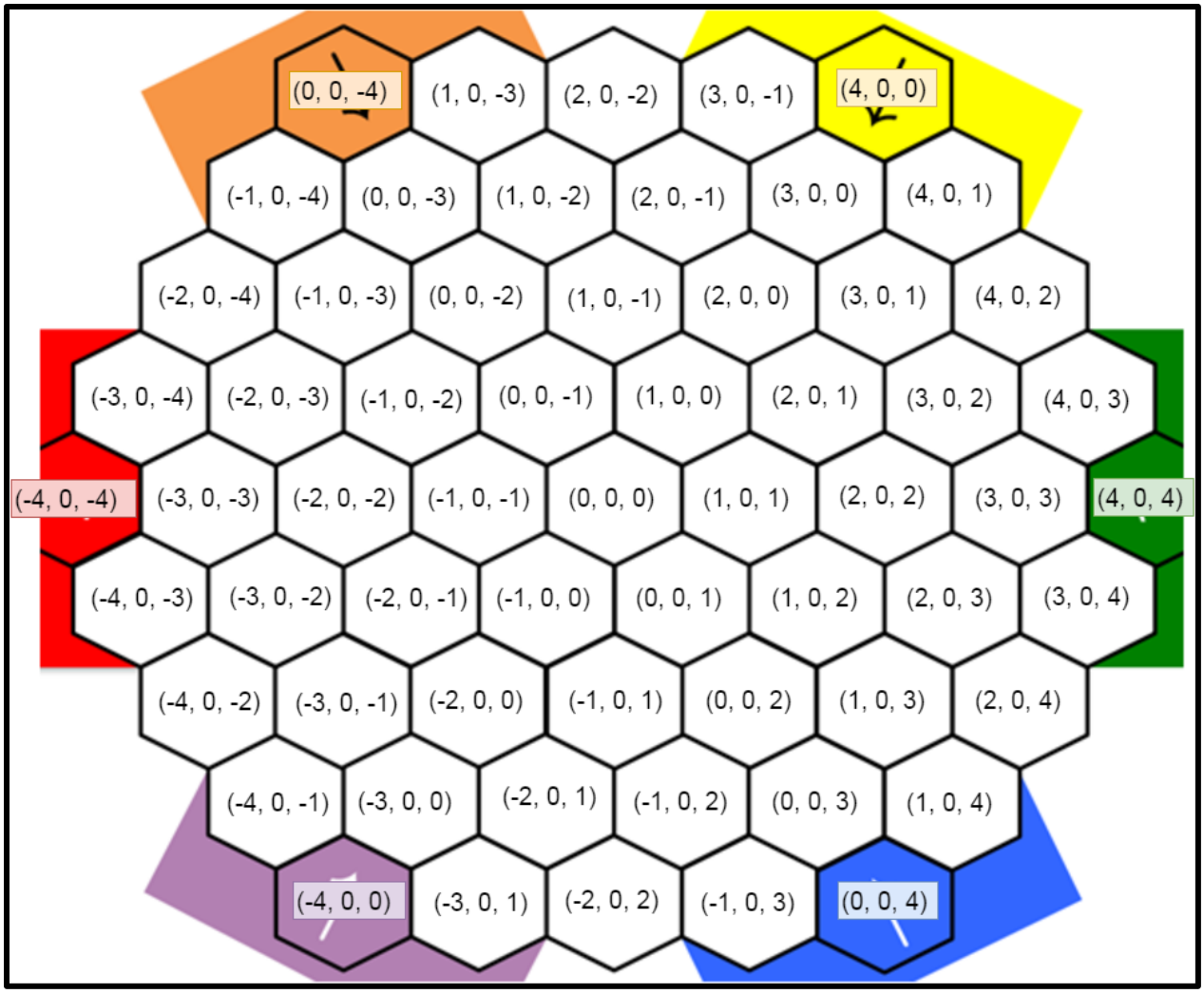


Figure 3: Reduced Coordinate Mapping

Although the reduced form saves us from having inconsistent names for a hexagon, it is not quite useful for measuring the shortest distance between two hexagons. It is required to find the smallest x, y and z to get to the hexagon from the origin. This is why there is an un-reduce function called toVector() function which rearranges the coordinate for the purposes of calculating shortest relative displacement between the origin and the coordinate. The unreduce() function makes use of the same fact that 1 unit in +x, and 1 unit in +z results in 1 unit in +y direction, and 1 unit in -x, and 1 unit in -z results in 1 unit in -y direction.

Calculating the Shortest Distance

To calculate the shortest distance between two hexagonal coordinates, the two essential methods required are shortestDistance() and displacement().

The shortestDistance Method

The shortestDistance method accepts two hexCoords. It first converts them to the reduced state.

To make the common case fast, it compares whether the two coordinates are aligned in x or z axis. If this is the case, then it just returns the distance between the two coordinates.

If however, this is not the case, the displacement method is called to calculate the shortest x, y and z lengths needed to go from the first coordinate to the second. The x, y, and z lengths are then added together and returned as the shortest distance.

The displacement Method

The displacement method accepts two hexCoords. It first calculates the smallest x,y, and z needed to get to each coordinate individually from the origin using the toVector() method. It then calculates the x,y and z distance between the two coordinates.

Now, because of the fact that (1,0,1) ←→ (0, 1, 0) and (-1,0, -1) ←→ (0, -1, 0), the method calculates the minimum x, y and z displacement between the two coordinates and returns the vector between the two coordinates. This vector stores information about how many minimum units of x, y and z are needed to get from the first coordinate to the second coordinate.

The Display Component

The **Robo-Wars** system, although straightforward with respect to most of its code and logic, contains some difficult code and logic within the system. To allow for future maintenance of the system, necessary classes of the **Display component** as well as their fields and methods are described here. Note that not all classes of the **Display component** and their respective functionality are documented because most of the code and logic does not require description.

All of the screen classes possess methods for constructing the screen itself, most of which are not described here as these were design decisions and none are beyond simple comprehension. Also, most objects within the system that are used to collect user input have been assigned abstract methods from the **Flow class**, none of which are beyond simple comprehension, but it is worth noting where these abstract methods are coming from.

Display Class

The **Display component** of the system has been implemented within one class as an extension of the Java JFrame and each screen stored and used by the **Display class** has been implemented as its own class as an extension from the **Screen class**, which is itself an extension of the JPanel class. Implementing each screen as its own class is necessary to separate the code and logic of each screen and to allow for modularity among the screens. This allows each screen to be focused on some small subset of the functionality of the **Display component**. The important fields and methods of the **Display class** are described below.

The screens within the **Display class** are stored as layers. The **Display class** assigns the Java layout CardLayout to the **Display class** to achieve the layering of the screens. When the **switchTo()** method is called, the visibility of the screens are adjusted so that the correct screen is visible.

Fields

- **screens**: The screens field stores all of the screens of the Display class within one array, which can be accessed with the index corresponding the ScreenEnum enumeration of each screen. The

array is an array of type Screen and stores each screen. The ScreenEnum enumeration is located within the ScreenEnum.java and simply enumerates the screens.

- **currentScreen**: The currentScreen field simply stored the ScreenEnum enumeration of the current screen that is being displayed. This field can be used to index the screens field to access the current screen that should be visible.

Methods

The **Display class** contains methods for initializing each of the screens stored by the **Display class** as well as some accessor methods, each of which are straightforward in logic and do not contain any difficult code. Only the important or difficult methods have been documented.

- **switchTo**(ScreenEnum screenEnum): The **switchTo** method takes one parameter, screenEnum, that represents the enumeration of the screen to switch to. The **switchTo** method switches the visible screen by setting the visibility of the previous screen to false so that it is no longer visible and then displays the next screen by setting the visibility of the specified screen to true so that it is now visible.

GameScreen Class

The **GameScreen class** has been implemented to represent the game screen within the **Display component**. The **GameScreen class** extends from the **Screen class** to obtain the basic methods needed for a screen. However, the **GameScreen class** implements most of its own functionality. Although the **GameScreen class** contains a large amount of code, most of it is not difficult and contains simple logic and reasoning. Most of the methods within the **GameScreen class** are implemented to simply pass parameters to the **board** field, so are not documented here. There are some fields and methods that proved difficult to implement and are documented for that very reason.

Fields

- **board**: The **board** field simply stores the game board as an instance of **BoardPanel**, which implements most of the functionality of the board. The **board** field is stored here to allow the rest of the system to interact with the **GameScreen** object and then the **GameScreen** object interacts with the **board** field to change the board as necessary.

Methods

- **movePiece**(int pieceOffset, HexCoord source, HexCoord vector, int range, HashMap<Integer, Integer> teamMembers, ArrayList<Integer> visibleRobots): The **movePiece()** method takes in the integer offset of the piece that is to be moved, the hexagon coordinate of the piece from which the piece is moving, a hexagon coordinate representing the vector of movements the piece will move, the range of the moving piece, a hash map of the moving piece's team members, and an array list of the visible robots in the order of description. The method then computes the list of movements the piece must be moved in order to animate the movement by adding a new hexagon coordinate to the list of movements for each x, y, and z integer within the vector parameter. The positive integers are positive movement and the negative integers are negative movement along the respective axis, and these values added or subtracted as necessary to derive the list of movements. After the list of movements have been computed, a new thread is created that loops over the list of movements, reduces the movements, resets the visibility of the board, resets the colors, shows the movement, and highlights the moved piece.

The thread then reloads the necessary visibility and redraws the game board. Finally, the thread sleeps for half of a second to allow the player to see the movement before continuing on to the next movement.

Note the **GameScreen class's createTopPanel()** (which is now on the left side) method is mostly trivial, but makes use of the **createPlayerPanel()** method which uses the **PlayerPanel class**. The **PlayerPanel class** is designed to allow the image and the attributes of the active player's active piece to be changed, which is done in the **setCurrentStats()** method. The **rotatePlayer()** method is meant to be used to rotate the players so that the active player is always on top. Further analysis of the classes and methods briefly described here should reveal all the information necessary to improve the functionality of the side players' bar.

BoardPanel Class

The **BoardPanel class** has been implemented to represent the game board that is displayed within the **GameScreen** object of the **Display component**. The **BoardPanel** class extends from the Java JPanel to allow the **GameScreen class** to store and manipulate the board as necessary. Extending from the Java JPanel class also enables the **BoardPanel class** to implement the **paintComponent()** method, which is used to draw the actual game board and game images. The **BoardPanel class** also contains private classes for simplifying the storage of the game images used. It is also worth noting the game images are loaded from the resources folder. The important fields and methods are documented to ease the task of understanding the **BoardPanel class**.

The hexagons stored within the **hexagons** field are stored such that the first layer, or first index, corresponds the first diagonal layer in the bottom left corner and then subsequent indices correspond to other layers in ascending order to the top right corner. More precisely, the first layer consists of the hexagons from the red home space to the purple home space, the middle layer consists of the orange home space to the blue home space, and the last layer consists of the yellow home space to the green home space. This ascending layer method of storing the hexagons is helpful in making use of the reduced method of the **HexCoord class** and also aids in the processing of the hexagons within the **hexagons** field since the reduced coordinates of the hexagons can then be mapped to indices.

Figure 4 below illustrates how the hexagons have been stored. The red line indicates the hexagons within the first layer, the orange line indicates the hexagons within the middle layer, and the yellow line indicates the hexagons in the final layer. Note this image is for a board of side length five and the indices will change when the board size length is increased to seven, but the ordering is the same. The top number on the hexagons is the first index used to store the hexagon in the **hexagons** field and the bottom number on the hexagons are the second index used to store the hexagon within the **hexagons** field.

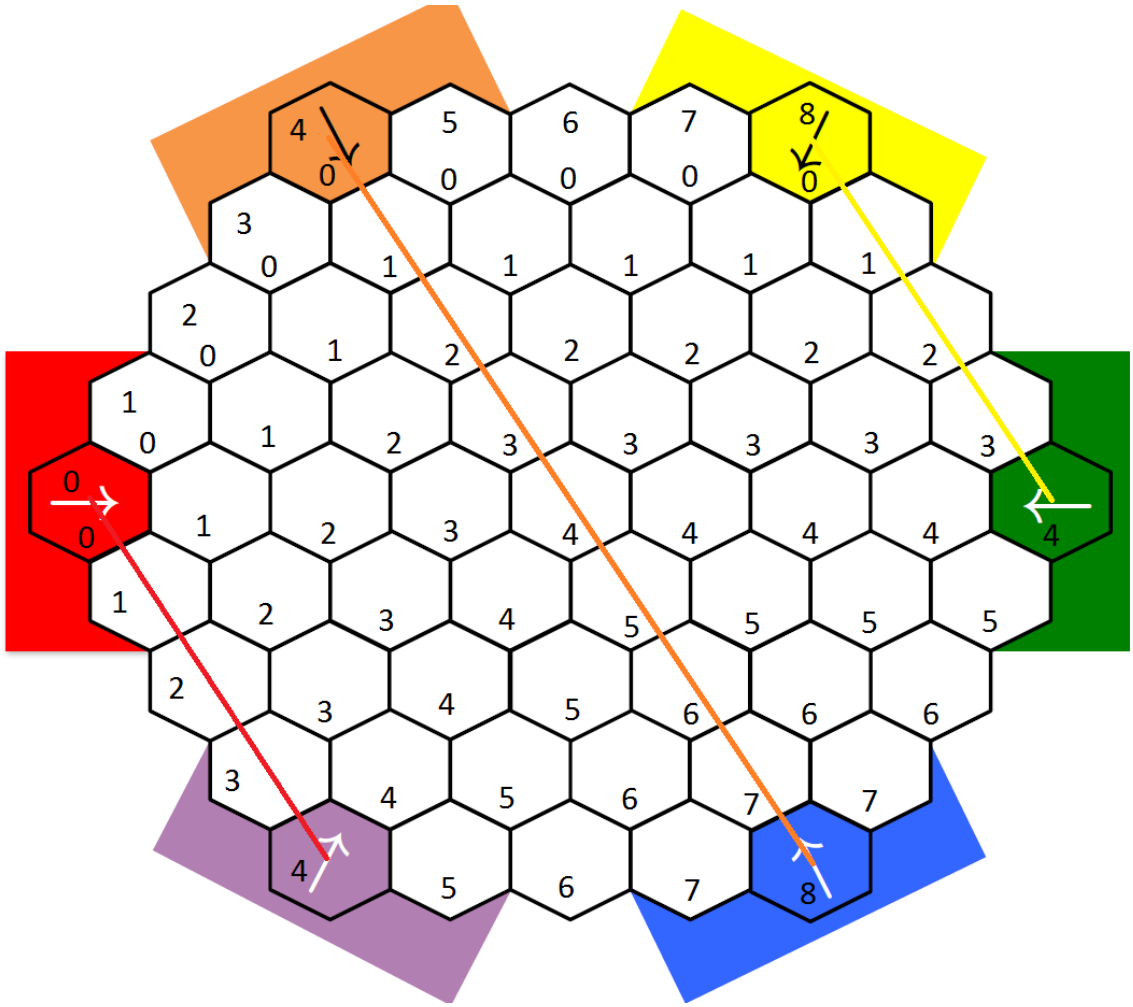


Figure 4: Hexagon Storage Schema

Note that most of the methods and fields within the **BoardPanel class** are trivial and only the nontrivial methods and fields of the **BoardPanel class** are described below as most of the trivial code relies on the nontrivial code described below.

Fields

- **hexagons**: The **hexagons** field is an array list storing the type **Hexagon**. The **hexagons** field is used to store the colors of the individual hexagons, the instructions on how to draw the hexagons, and the positions of the individual hexagons, all of which is stored in the **Hexagon class**.
- **boardSize**: The **boardSize** field is of type integer and stores the size of the board. This field is assigned in the constructor after receiving the board size as a parameter.
- **addFactor**: The **addFactor** field is of type integer and represents the factor that is used when mapping hexagon coordinates in the **hexagons** field. That is, the mapping of the first index in the **hexagons** field to the corresponding x coordinate requires adding a factor, which is always one less than the size of the board. The **addFactor** field also serves as the indicator of the center layer within the board, which is necessary to correctly draw the board, or the first index of the center layer within the **hexagons** field.

- **robotImages**: The **robotImages** field is an array of type **RobotImage** and stores the images of the robots that will be drawn onto the game board as well as their positions and instructions for drawing the images, all of which is contained in the private **RobotImage class.**
- **contextMenu**: The **contextMenu** field is of type JDialog and stores the context menu. The context menu if the menu of buttons that appears when a player clicks on the game screen.
- **shot**: The **shot** field is of type **ShotImage** and stores the images of the shot animation, the positions of each of the shot animation images, and instructions on how to draw the images, all of which is contained in the private **ShotImage class**.
- **bangVisible**: The **bangVisible** field is of type boolean and stores true if the image representing the "bang" of the shot is visible and false otherwise.
- **mushroomVisible**: The **mushroomVisible** fields is of type boolean and stores true if the image representing the mushroom cloud is visible and false otherwise.

Methods

- **inititalizeHexagons**(Point center, int numberPlayers): **inititalizeHexagons()** takes in two parameters, the center point around which the board will be positioned and drawn and the number of players. The coordinates at which to begin initializing the images are then determined by subtracting the **addFactor** multiplied by the width of the hexagons and then multiplying by the square root of three, which is done because the width of each hexagon is the length from the center of the hexagon to any of the six points where the sides lengths connect. The square root of three can be determined by analyzing the angles and positions of the hexagon side lengths and the points at which each connect. This computation determines the beginning drawing point from the bottom left corner of the hexagon grid that represents the game board. Next, the **inititalizeHexagons()** method initializes the first half of the hexagons of the board by using two for loops to loop over each hexagon according to the board size and the increasing number of hexagons as the center approaches and continuously adding to the starting x coordinate and subtracting from the starting y coordinate to position each hexagon. Finally, the other half of the hexagons are drawn in a similar fashion, but with modifications to the loop for assigning hexagon positions. The hexagon colors are assigned depending on the number of players and each of the loop counters because the number of players dictate the colors of the hexagons and the loop counters determine which hexagon is being drawn. This portion of the code was merely a matter of determining when to assign a color to a hexagon and which hexagon to assign it to, which is determined from the game board specification. Finally, note the condition to change the positioning of the hexagons depending on the board size when initializing the upper portion of the game board is necessary to correctly initialize the hexagons, but it is not understood why.
- **initializePositions**(int numberPlayers): The **initializePositions()** method initializes the positions of each piece image and takes in one parameter, the number of players. It is important to note the **initializePositions()** method must be called after the **inititalizeHexagons()** method as it relies on the colors of the hexagons to correctly compute the initial positions of the piece images. The piece images are initialized by looping over the number of players in a for loop as well as looping over every hexagon within the outer for loop that loops over the number of players. The initial positions of the images are then assigned based on the color of the hexagon, which is why it is essential the **inititalizeHexagons()** is called prior to this method. For each of

the hexagons, if the color matches one of the team colors and the required number of players for that color to be active are met, the relative position of that team's pieces within the **robotImages** field is determined and the initial position of each of the team's piece images for the current hexagon color are set to the position of the hexagon that has been determined to have the team color.

- **reducedHexToIndices**(HexCoord coord): The **reducedHexToIndices()** method takes in one parameter, a hexagon coordinate, and uses it to determine the indices of the hexagon within the **hexagons** field corresponding to that hexagon coordinate. The **reducedHexToIndices()** computes the indices of the hexagon represented by the hexagon coordinate based on the fact that the stored hexagon coordinates increase in their x coordinate value as the first index of the **hexagons** field index also increases and the z coordinate increases as the second index increases. So the index can be computed by adding the **addFactor** field to both the x and z coordinates of the passed hexagon coordinate. The condition that changes the second index when the computed first index is greater than the **addFactor** field is to correct the index since the z coordinates begin increasing along with the x coordinate (and also the first index) after the center layer. The second index is computed by subtracting the difference between the first index and the **addFactor** field.
- **indicesToReducedHex**(Point point): The **indicesToReducedHex()** method takes in one parameter, the indices of a hexagon as a point, and uses it to compute the reduced hexagon coordinate of the corresponding hexagon. The logic is the reverse of the **reducedHexToIndices()** method.
- **getClickedHexagonCoords**(Point mousePoint): The **getClickedHexagonCoords()** method takes in one parameter, the point of the mouse, and uses it to determine the hexagon coordinates of the hexagon that was clicked. The **getClickedHexagonCoords()** method determines the hexagon that was clicked by looping through the hexagons stored in the **hexagons** field and computing the distance between the mousePoint parameter and the center position of each hexagon. If the distance is less than or equal to the width of the hexagons, then the current hexagon within the loop is determined to be the clicked hexagon. The coordinate of the hexagon is then computed similar to how the **indicesToReducedHex()** method determines the hexagon coordinate.
- **getClickedHexagonIndices**(Point mousePoint): The **getClickedHexagonIndices()** method takes in one parameter, the point of the mouse, and uses it to determine the indices of the clicked hexagon. The **getClickedHexagonIndices()** method computes which hexagon was clicked similar to the **getClickedHexagonCoords()** method, but computes the indices similar to the **reducedHexToIndices()** method instead of computing the hexagon coordinates.
- **getHexagonsInRange**(Point indices, int range): The **getHexagonsInRange()** method takes in two parameters, a pair of indices of a hexagon stored in the **hexagons** field and the range in which to retrieve hexagons, and uses them to compute a list of hexagons within the specified range of the specified hexagon. The **getHexagonsInRange()** method computes the list of hexagons within the specified range of the specified hexagon by first determining the hexagons that are in range above the specified hexagon including the layer in which the specified hexagon resides and then computes the hexagons within range below the specified hexagon. A hexagon is determined to be in range in the same layer by simply including the hexagons whose difference in the first index relative to the specified hexagon is less than or equal to the specified range. The range is then decreased by one and the next layer is examined to determine other hexagons in range.

The entire collection of hexagons within the specified range of the specified hexagon are computed in this fashion from the middle outwards. The idea is to "squeeze" the area above and below the specified hexagon to only include the hexagons in range.

- **paintComponent**(Graphics g): The only aspect of the **paintComonent()** method that is nontrivial is how the robot images are drawn. The robot images are drawn by determining if any of the robots are occupying the hexagon space that is currently being drawn within the loop the redraws the hexagons on the game board. If a robot is occupying a hexagon space that is visible, then the robot image for that robot is drawn. The **isOccupying()** method determines if a robot image is occupying a space by computing the distance between the center of the hexagon and the center of the robot image. If this distance is within the width of the hexagon, the robot image is determined to be occupying that hexagon. Finally, the visible hexagons are the ones that are visible to the currently active piece or any of its team members, so if a robot is occupying a visible hexagon, it is drawn while looping through the hexagons in the **hexagons** field and redrawing the game board.

Private Classes

- **RobotImage**: The **RobotImage class** is designed to model the object that will store the collection of data related to each robot image to ease the manipulation of the robot images. As such, the **RobotImage class** has been designed to store the image of the robot, the instructions for drawing the robot image, the coordinates of the images position on the screen, and accessor and mutators for these attributes.
- **ShotImage**: The **ShotImage class** is designed similar to the **RobotImage class**. It is designed to store the images to animate the shot, the positions of the images, instructions to draw each image, and has accessors and mutators for each of its attributes.

Hexagon Class

The **Hexagon class** has been implemented to represent a hexagon space on the game board. It is used within the **BoardPanel class** to draw the hexagon game board. Most of the **Hexagon class** is simple and contains multiple fields and methods pertaining to its position, color, and visibility. Only the **drawComponent** is documented because it is the only aspect of the **Hexagon class** deemed requiring some description.

- **drawComponent**(Graphics2D g2): The **drawComponent()** method takes in one parameter, the 2D graphics object that will be used to draw the hexagon. The hexagon is then drawn using the graphics object it received by instantiating a polygon, computing its six points based on the center coordinate of the hexagon, drawing the outline of the hexagon in black, and finally filling the hexagon with white.

RobotArchiveScreen Class

Most of the **RobotArchiveScreen class** is trivial, but it does merit some description to understand how it functions so that it may be completed in the future. The labels on the right of the screen that display the statistics of the robots are stored within the **statsLabel** field. These labels are used within the **setStats()** method to set the displayed stats of the robot. The enumerations of the robot files were to be displayed within the JPanel stored by the **textPane** field, which is stored within a Java JScrollPane to allow the JPanel to be scrolled when there are large amounts of enumerated robot files.

Finally, the **addEnumeration()** method adds an enumeration as an instance of the **RobotLabel class**, which has been designed specifically for the enumerations of the robot files to allow the selected **robotLabel** instance to be highlighted, to the listed enumerations within the **textPane** field.

ResultsScreen Class

Again, most of the **ResultsScreen class** is trivial, but it does merit some discussion. The **ResultsScreen class** has been designed to be constructed only once. It has been designed to instantiate a Java JTabbedPane that will be used to store the colored panels storing the results of each team that participated within the match. The **addResults()** method is used to add the results of a player. Note the **ResultsScreen class** needs to be improved to only insert data for teams that participated within the game, not all six players.

PlayerSelectionScreen Class

The **PlayerSelectionScreen class** merits discussion for the amount of code present. The construction methods of the **PlayerSelectionScreen class** are trivial and can be understood by simply analyzing the code. However, any programmer should know the **PlayerSelectionScreen class** stores fields for each of the radio buttons used to select the number of players and uses these fields to determine whether to display the Start Game button and the Board Size drop-down menu. The **playerSelectAIs** and the **playerPanels** fields are used to determine which players are AIs and their entered names. Also, the **playerSelectAIs** should allow the user to select the AI to play against, but does not. Further work is required to enable the selecting of AI files and enabling the options to actually impact the system when visiting the Game Options screen through Game Options button.

Interpreter Component

Within the game system, the Interpreter component is responsible for parsing the programs of AI robots. These robots are files externally created and encoded as JSON files.

JSON Library Information

AI robot scripts are held as JSON files and are then read into the system using the following external libraries.

JSON.simple

JSON.simple is used as an external Java library for processing JSON in this project. The library file is named as json-simple-1.1.1.jar and it is located at the "Referenced Libraries" folder. This is mainly used in the RecordLoader.java in the "robowars.flow" package to read/update/parse robot information which is in the "resources/ExampleRobots" folder. There is a detailed wiki page on Google Code: https://code.google.com/archive/p/json-simple/wikis.

Main packages: "org.json.simple" and "org.json.simple.parser"

First, every *.jsn file will be read from normal Java FileReader, then it will be converted to JSONObject by JSONParser. JSONObject is a special class inherited from java.util.HashMap, so it is also a key-value pair data structure and has the methods that HashMap has. In this case, all the *.jsn files are starting with "script" key, so just get the value of "script" key as main JSONObject for later usage.

Different attributes (team, name and class) and statistics (matches, wins, losses, executions, lived, died, absorbed, killed and moved) are separated from that main JSONObject. In addition, the value of "code"

key for the interpreter is specially handled as JSONArray since it contains an array of strings. JSONArray is inherited from java.util.ArrayList, so it has methods that inherited from AbstractList, ArrayList, List and so on.

Basic usages: Parse/Read/Update JSON

To parse a JSON file you invoke the method parser.parse(FileReader). If you wish to read a JSON file, you invoke the method JSONObject.get(key). Finally, if you want to update a JSON file, you invoke the method JSONObject.put(key, new_value)

Known Issues with Robot Library

Ideally, the system would draw these from an online collection of robot records, but the software as deployed can only retrieve robots which are stored locally, in the `/resources/ExampleRobots` folder. Some of these robots were discovered to not follow the specifications for the robot language, and only the robot `0FixedCentralizer.jsn` has been confirmed to work properly. However, the Interpreter should be capable of parsing and playing any correctly formatted robot script.

Script Representation

Each AI program is loaded as a list of strings representing each line of the program, and when an AI player is initialized, these strings are stripped of their comments, whitespace removed, and then appended to a clean list as sanitized program data. When a match begins, the pieces for each robot player are initialized with their respective programs. These programs are run from start to finish in an initialization mode, where certain built-in functions of the parser (such as moving a robot or shooting) are disabled. This has the effect of creating and populating any program variables and defining new functions ("words"). Each robot program requires a `play` word to be defined to be run properly; if a program does not define one, the Interpreter creates a blank one.

Every operation performed by the Interpreter on behalf of a program involves either the list of current instructions, or a stack used for operations. When a piece's turn is called, the `play` word is placed on the list of current instructions, and the parser called. The parser replaces the `play` word with its code (defined during initialization), and then the parser will continue to run until the list of current instructions is empty, upon which the Interpreter will end the piece's turn.

The parse() Method

The parse() method is the core of the Interpreter. It examines each term of the list of current instructions and performs the appropriate action on it after determining its nature.

- If the term is an empty string, the parser will do nothing.
- If the term is a semicolon, the parser will clear all loop iterators and reset the semicolon flag.
- If the term is "`I`", the parser will attempt to retrieve the value of the current loop's iterator and place it on the stack, ending the AI's turn and throwing a debug error message if no iterator exists.
- Then parser searches the Interpreter's list of standard built-in functions (43 in all, including arithmetic operations and board queries) to see if the term matches any of these. If so, it will call the built-in function using the Interpreter's function table. Regardless of whether the called

operation succeeds or fails, the parser returns normally. If the mode is illegal (trying to call restricted operations like shoot() during initialization), it will throw an error message, clear the current instruction list and exit.

- Next, the parser searches the list of user-defined words for this program to see if the term matches. If it finds a match, it appends the replacement instructions for that word to the beginning of the current instruction list.
- The parser then searches the list of user-defined variables for this program to see if the term matches. If it finds a match, it pushes the "address" of the variable to the stack by appending a # symbol to the term and pushing it to the stack.
- If none of the above are true, the term is a value that should be pushed to the stack. The parser checks if this term begins with a "." Character, signifying a string, and if it is a string, it strips out the dot and quotation marks before pushing the value to the stack.

No-Tolerance Policy

In order to protect the user against the adverse effects of malformed or malicious user-created programs, the Interpreter was created with a no-tolerance policy for failing code. Since many operations involve the stack, if one operation has incorrect parameters, it is assumed that it will not leave the stack in an appropriate condition for the following operation, having a domino effect on all operations afterwards. Therefore, if an operation fails for not having correct arguments or if some other error state occurs, the Interpreter immediately clears the current instruction list and the stack, which has the effect of stopping the AI's turn after that operation.

Additionally, the Interpreter runs a timer on any loop-related operations, in order to prevent infinite loops from locking up the software or causing delays. This timer is set at the beginning of loop operations, and if the timer expires, the AI's turn ends immediately.

The Logger Component

The Logger feature was advertised as a "Might-have" in the Requirements document. The purpose of this feature was to log events in the game and display it to the players who can witness the events.

Known Issues

Although the Logger component has been implemented as well as integrated with the Board, there still exist minor bugs in the classes, such as the displayShooting() method in Logger.java. The displayShooting() method in Logger.java throws an Index out of Bounds exception, when called. Also, the hexagonal coordinate positions of the pieces are not always correct, and need to be fixed.

Integration Issues

Furthermore, the logger component has not yet been integrated with the Display component. It was planned that the logger would have its own window at the bottom left of the game screen, with events displayed for the player team in the hot-seat that the player's pieces should have seen, based on their range.

Extensions to the Software

During development, some pieces of integration were not completed and some requirements were not met. However, scaffolding exists within the system that will allow for easy implementation of several extra features in the future.

One possible nice-to-have feature documented during the requirements stage was the implementation of an “advanced” ruleset. This would include allowing for a variable number of pieces per team (eg. 2 snipers and 3 tanks per team), and custom maps to play on with different board sizes and “blind” spaces. These components have actually been incorporated into the game logic, and are missing in the final system only because they lack an interface to set their parameters with.

No part of the game logic stored in the board assumes a hard-coded number of pieces per team, and a match can be constructed with a variable number of pieces per team. Any programmer wishing to incorporate this functionality would need to add these parameters to the Game Options screen and connect the GUI elements to their values.

The system has also been constructed to take a map name when creating a match, which can be used to retrieve hardcoded “boundary” hexes for each map.