# 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 Event Handler 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 Event Handler 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.

# 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

### 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.

## 2.2 Flow Component Classes

The Flow component regulates the flow of the game based on user and player input. The Flow component is essentially divided into two sub-components, EventCatcher and MenuManager. The EventCatcher c

The MenuManager sub-component is designed for the events initiated by the User, which is outside of the match and the EventCatcher sub-component is designed for all Player responses.

The Flow 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 eventDispatched(AWTEvent) function is the hub of all events and it is located in the main Flow class. It calls the delegate functions to perform the specific job based on the particular event. This function accepts the AWTEvent as a parameter.

Each of the MenuManger and EventCatcher functions contain an array of Strings which stores all the possible button names in their scope of screens.

A private function IsValidButton() is used to check whether or not any of the arguments of button names are valid.

The function clickedButton(String ) is located both in MenuManager class and EventCatcher class. The function in the MenuManager deals with the buttons clicked on all screens other than the game screen. The function in the EventCatcher handles the buttons on the game screen. Both the functions take in the name of the button which is clicked as a parameter, throw an exception if the button name is invalid, and return nothing.

The function clickedEndTurn() is called when the Player decides to end their turn. The function neither accepts any parameters nor returns a value.

The function clickedMove(int ) is called when the Player decides to move their piece to a position. This void-returning function accepts an integer variable which refers to the position at which the piece wants to move.

The function clickedShoot(int ) is called when the Player decides to shoot at a piece at a position. The function accepts an integer variable which refers to the position at which the piece wants to shoot and does not return any value.

The void-returning function clickedExitScreen() is called when the Player wants to quit the Match. This function does not accept any parameters.

The function clickedPanArrow() is called when the Player chooses to switch between the Pan mode and the Arrow mode.

The draggedGameBoard(mousePosition pre, mousePosition post ) function is called when the player is panning the game board. This is a void-returning function and it accepts initial and final positions of the mouse.

## 2.3 Board Component Classes

## 2.4 Pieces Component Classes

## 2.5 Interpreter Component Classes

## 2.6 Logger Component Classes

### 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 HexCood 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 HexCood of the target location, a TeamEnum and a PieceEnum representing the piece from one team shooting a target location.

The setDamaged() method takes a HexCood, 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.

## 2.7 The Complete Class Structure

# 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.