VOOGASalad Application Program Interface (API)
Group: VOOGirlsGeneration

Authoring Environment

AUTHORING ENVIRONMENT DATA

```
/**
 * Interface for data stored in the authoring environment to allow for ease of saving into JSON
 * @param <T> - Type of object to be stored in data
 */
public interface AuthoringData<T> {

	/**
	 * Adds a new object of type T to data
	 */
	public void add(T arg);

	/**
	 * Removes an object of type T from data
	 */
	public void remove(T arg);

	/**
	 * Clears the data
	 */
	public void clear();
```

GAME COMPONENT CREATOR DIALOGS

```
/**
 * Opens a dialog for creating new Actions in the authoring environment
 */
public class ActionCreator extends PopupWindow

	/**
	 * Constructor for an ActionCreator popup window
	 * @param actionData - class where user-created Actions are stored
	 */
	public ActionCreator(ActionData actionData)

/**
 * GUI element used to create new Patch objects and add them to the library. Allows users
 * to specify the name and image of the patch.
 */
public class **PatchCreator** extends PopupWindow
        /**
         * Constructor that sets the dimensions of the PatchCreator GUI component
         * and initializes it.
         * @param patchData - class where user-created patches are stored
         */
        public **PatchCreator**(PatchData patchData){

/**
 * GUI element that allows users to create new Piece templates and add them to the
 * Library. User defines unit name, image, and actions. Actions define a units behavior
 * and ultimately make the unit what it is.
 */
public class **PieceCreator** extends PopupWindow

   /**
    * Constructor that sets the dimensions of the PieceCreator GUI component and initializes it.
    * @param piecesData - class where user-created pieces are stored
    */
   public **PieceCreator** (pieceData pieceData);

/**
 * Popup GUI element that allows the user to specify the size of the grid, selects the
 * tiles and returns the list of relative coordination of selected tiles.
 */
public class RangeEditor extends PopupWindow
```

VIEW

```
        /**
         * The GUI contains all the parts in authoring environment. It sets the size
         * and the position of the parts in the GUI.
         */
        public class VoogaView extends BorderPane
```

GRID VIEW

```
/**
 * The GUI components for the grid displayed on the right side of the game authoring
 * environment. It displays all the unit/terrain which are chose to put on it. It also
 * demonstrates currently selected tile. It scrolls when the size of the grid exceeds a certain size.
 */
public class GridView extends ScrollPane
        /**
         * Get the grid which is the content of the GridView.
         * @return Grid which contains all the tiles.
         */
        public Grid getGrid()

/**
 * The grid which contains all the tiles and draws the tiles and grid lines.
 */
public class Grid extends Pane
        /**
         * Get the number of tiles in a row.
         * @return The number of tiles in a horizontal line of the grid.
         */
        public int getGridWidth()

         /**
         * Get the number of tiles in a column.
         * @return The number of tiles in a vertical line of the grid.
         */
        public int getGridHeight()

         /**
         * Get a specific tile in the grid according to its position.
         * @param x: The X coordination of the tile
         *                      from the left smallest to the right largest.
         * @param y: The Y coordination of the tile
         *                      from the bottom smallest to the top largest.
         * @return The tile at the specified position.
         */
        public Tile getTile(int x, int y)

/**
 * The view of the grid especially for selecting the range.
 */
public class RangeGrid extends GridView
        /**
         * Update the grid with new number of tiles in rows and columns from user type in.
         * Demonstrate the selected tiles and set a image in the center of the grid.
         * @param widthGridNumber: The number of tiles in a row.
         * @param heightGridNumber: The number of tiles in a column.
         * @param myTileSize: The preferred size of the tile.
         */
        public void update(int widthGridNumber,int heightGridNumber,int myTileSize)

        /**
         * Add the image at the center of the grid as a reference.
         * @param image: The image to put in the center.
         */
        public void addCenterImage(Image image)

        /**
         * Collect all the coordination of selected tiles relative to the center tile
         * as Point2D in a list.
         * @return The list of relative coordination relative to the center tile.
         */
        public List<Point2D> getSelectedList()
```

<u>LIBRARY VIEW (VIEW OF LISTS OF GAME COMPONENTS)</u>

```
/**
 * GUI components for the library displayed on the left side of the
 * game authoring environment, which contains all instantiated units
 * and terrain. From here, the user can open the Unit/TerrainEditors
 * to edit the units and terrain, as well as select them for
 * placement on the grid.
 */
public class LibraryView extends TabPane

        /**
         * Method to add units and terrain to their respective tabs in the LibraryView.
         * @param content : The LibraryEntry to be added to the library.
         * @param library : Specifies whether to add the content to the
         * "Unit Library" or the "Terrain Library".
         */
        public void addToLibrary(HBox content, String library);

/**
 * An HBox containing data for a unit piece. PieceEntries are
 * added to the LibraryView in the game authoring environment.
 */
```

```
public class **PieceEntry** extends LibraryEntry
	/**
	 * Method to retrieve the Piece class contained within this PieceEntry class.
	 * @return Piece associated with the PieceEntry.
	 */
	public Piece **getPiece**();

/**
 * An HBox containing data for a terrain patch. PatchEntries are
 * added to the LibraryView in the game authoring environment.
 */
public class **PatchEntry** extends LibraryEntry
	/**
	 * Method to retrieve the Patch class contained within this PatchEntry class.
	 * @return Patch associated with the PatchEntry.
	 */
	public Patch **getPatch**();
```

Game Player

InitialScene View____

```
/**
     * the method allows user to load the previously saved json representation
     * of the game and uses JSON reader from Game Data to generate an instance
     * of Game.
     */
protected void loadGame ()

    /**
     * generates drop down menu that allow user to choose a new Game to play
     * The Games are generated from the directory that stores all json files defined
     * from authoring environment
     */
protected void newGame ()
```

Game Space View____

```
/**
* the method to restart the game; it shows a pop up diaglogue and asks the user whether to
* save the current game
*/
protected void restartGame ()

 /**
  * exits the game; the stage closes upon on click
  */
protected void exitGame ()

    /**
     * to save the current game (state and settings). write to a json file which could be later loaded in
     */
protected void saveGame ()

    /**
     * Update the stats panel with stats from the selected piece
     * @param piece the selected piece upon on click
     */
protected void updateStats(Piece piece)

    /**
     * Update the action panel with actions of the selected piece
     * @param piece
     */
protected void updateActions (Piece piece)

/**
     * Movement map (which maps movements to keycodes) is passed by the game engine.
     * This method creates key event handlers so that when such a keycode is
     * pressed, its corresponding movement is implemented. Movement is "moving the
     * cursor" with the keyboard.
     *
     * @param movementKeyMap
     * @param gameScene
     */
    public void setMovementKeyControl(Map<KeyCode, Point2D> movementKeyMap, Scene
gameScene)

    /**
     *
     * Action map (which maps Actions to KeyCodes) is passed by the game engine.
     * This method creates key event handlers so that when such a keycode is
     * pressed, its corresponding action is implemented.
     *
     * @param actionKeyMap maps actions to keycodes
     * @param gameScene is the scene for GUI
     */
public void setActionKeyControl(Map<KeyCode, Action> actionKeyMap, Scene gameScene)
```

Score Board View:

```
   /**
    * loads the players and their scores of the current game;
    * display the Highest score in the high score display at the bottom
    */
protected void loadScores()
```

Game Engine

GAME ENGINE

```
/**
 * Initialize a GameLoop, JSONParser and prepares the engine for game loading
 */
public class GameEngine
        public GameEngine()
```

GAMELOOP

```
/**
 * Main GameLoop of the Game Engine, plays the game
 */
public class GameLoop
        public GameLoop()

        /**
         * Called whenever a change happens in the game state
         * (player makes a move/behavior is executed)
         */
        public void getInput()

        /**
         * Executes rules and behaviors as a result of the state change
         * and updates the state of the game with it’s new state
         */
        public void processInput()

        public void updatePieces()

        public void updateGrid()

        /**
         * Returns preference file containing preferences for the game
         */
        public preference getPreferences()

        /**
         * Sets preferences for the game
         */
        public void setPreferences()
```

```
    /**
     * Writes the current state of the game to a JSON file
     */
    public JSON writeState()

    /**
     * Sets the current state of the game from a JSON file
     */
    public void setState()
```

PLAYER

```
/**
 * A player object that contains the logic for playing each level. This object
 * requires no parameters for initialization.
 */
public class Player
    /**
     * Default constructor
     */
    public Player();

    /**
     * Constructs a player with a specific ID
     * @param id int ID corresponding to the Player
     */
    public Player(int id);

    /**
     * Resets number of moves player for the player
     */
    public void resetMovesPlayed();

    /**
     * Getter to return the ID of the player
     * @return int ID of the player
     */
    public int getID();

    /**
     * used by game player (GUI) so that it knows what action to perform when
     * certain keycodes are pressed/used.
     * @return myKeyMap which maps actions to pre-defined keycodes
     */
```

```
public Map<KeyCode, Action> getActionKeyMap();

/**
 * need to get info from the authoring environment to set up the map.
 * @param myKeyMap
 */
public void setActionKeyMap(Map<KeyCode, Action> myActionKeyMap);

/**
 * Returns the Key Mapping for the Player
 */
public Map<KeyCode, Point2D> getMovementKeyMap();

/**
 * Sets the Key Mapping for the Player
 * @param myMovementKeyMap2
 */
public void setMovementKeyMap(Map<KeyCode, Point2D> myMovementKeyMap2);
```

Game Data

ACTION

```
/**
 * Interface of a game component that performs an action on another component in the game.
 * Pieces will contain a list of actions.
 */
public interface Action
        /**
         * Returns the name of the Action for display
         * @return name of Action
         */
        public String toString();

        /**
         * Gives back a list of Point2D of absolute locations for the action range
         * @return list of absolute locations in Point2D
         */
        public List<Point2D> getActionRange(Point2D pieceLocation);

        /**
         * Gives back a list of Point2D of relative locations for the effect range of the action
         *(splashzone)
         * @return list of relative locations in Point2D
         */
        public List<Point2D> getEffectRange();

        /**
         * Executes an action on a component of the game (i.e. a piece, patch, or other module)
         */
        public void doBehavior(Piece actor, Piece... receivers);

/**
 * Conclusion that runs at the end of each action.
 * Could be piece removal, more modifying of stats, etc.
 * Will be pre-coded in code and chosen by the user.
 */
public interface ActionConclusion

        /* Action conclusion constructor
         * @param actor - Piece that calls action
         * @param receivers - Pieces that receive the action
```

```
            */
           public void **runConclusion**(Piece actor, Piece... receivers);

/**
 * A concrete instance of an Action. All Actions defined by the user will be an instance of this class.
 */
public class **ConcreteAction** implements Action
           /**
            * ConcreteAction constructor is called when a new Action is made and
            * its behavior is already defined
            */
           public **ConcreteAction**(String name, List<Point2D> attackRange,
                         List<Point2D> effectRange, List<StatsTotalLogic> statsLogics,
                         ActionConclusion conclusion);

/**
 * Stores the overall logic for one stats modifying equation.
 * Note, a StatsModifyingAction contains a list of StatsTotalLogics.
 */
public class **StatsTotalLogic** extends StatsModifier
           /**
            * Constructor for StatsTotalLogic
            * @param target - One of 2 string choices indicating whether the stat to be
            * affected is that from the actor or the receiver. String choices: [actor, receiver]
            * @param stat - String name of stat to be modified
            * @param logic - List of StatsSingleMultipliers to edit the stat
            */
           public **StatsTotalLogic**(String target, String stat, List<StatsSingleMultiplier> logic);

/**
 * Stores one multiplication equation for the StatsModifyingAction.
 * Note, each StatsTotalLogic contains a list of StatsSingleMultipliers.
 */
public class **StatsSingleMultiplier** extends StatsModifier

           /**
            * Constructor for StatsSingleMultiplier
            * @param modifier - double containing scale factor of stat
            * @param target - One of 3 string choices indicating whether the stat to be
            * affected is that from the actor or the receiver, or if the value is a constant.
            * String choices: [actor, receiver, constant]
            * @param stat - String name of stat to be modified, or double as a string if for constant
            */
```

public **StatsSingleMultiplier**(double modifier, String target, String stat);

GAME

//Represents a Game
public class **Game**

public **Game()**
public **Game(**List<Player> players, List<Level> levels)
//Construct to build a game with Players and Levels
public void **play**()
//Rotates through player turns
public Level **getCurrentLevel**()
//Getter to return the current level being played in the game

GOALS

/**
 * A goal defines the win conditions for each level.
 */
public class **Goal**

public **Goal** ()
public int **checkGameState** (Level l);
//Checks to see if the Goal condition has been satisfied or not

GRID

/**
 * Contains the Grid defined for a level. Contains the pieces and patches
 */
public abstract class **Grid** {

/**
 * Default constructor for square grid
 */
public **Grid ();**

/**
 * constructor of grid
 * @param x number of rows
 * @param y number of columns
 */
public **Grid** (int row, int column);

/**
 * set up grid by initializing patches on it
 */
public abstract void **setGrid** ();

```
/**
 * places a patch on the grid
 * @param patch to be put on grid
 * @param coord of patch
 */
public void setPatch (Patch patch, Point2D coord) {
}

/**
 * gets the patch on the given coordinate
 * @param coord of patch
 * @return patch
 */
public Patch getPatch (Point2D coord) {
  }

/**
 * gets the piece on the given coordinate
 * @param coord of piece
 * @return piece
 */
public Piece getPiece (Point2D coord) {
}

/**
 * removes the piece on the given coordinate
 * @param coord of piece
 */
public void removePiece (Point2D coord);

/**
 * removes the patch on the given coordinate
 * @param coord for removal
 */
public void removePatch (Point2D coord);

/**
 * Returns number of columns in grid
 */
public int getColumn () ;

/**
```

```
     * Returns number of rows in grid
     */
    public int getRow ();

    /**
     * Returns a Piece of a given ID
     * @param id
     * @return
     */
    public Piece getPiece (int id);

    /**
     * @return a list of all pieces
     */
    public List<Piece> getAllPieces ();

    /**
     * @return  a list of all patches
     */
    public List<Patch> getAllPatches ();

    /**
     * gets the patch on the given coordinate
     * @param coord of patch
     * @return patch
     */
    public Map<Point2D, Patch> getPatches ();

    /**
     * gets the piece on the given coordinate
     * @param coord of piece
     * @return piece
     */
    public Map<Point2D, Piece> getPieces ();
```

INVENTORY

```
/**
 * Inventory to be contained by a piece if the user chooses to add an inventory to a piece.
 * Inventory contains a list of pieces.
 */
public class Inventory {
        private List<Piece> myInventory;
        /**
         * Constructor for inventory, initializes an empty inventory
         */
        public Inventory()

        /**
         * Adds the indicated pieces to the inventory
         * @param items - pieces to be added
         */
        public void addItem(Piece item)

        /**
         * Removes the indicated pieces from the inventory
         * @param items - pieces to be removed
         */
        public void removeItem(Piece item)

        /**
         * Set the inventory to an already filled list of pieces
         * @param items - pieces in the inventory given as a list of pieces
         */
        public void setInventory(List<Piece> items);

        /**
         * Checks whether inventory is empty or not
         */
        public boolean isEmpty();

        /**
         * Returns the actions contained in all the items as a list of Actions
         * @return list of Actions contained in the items
         */
        public List<Action> getItemActions();
```

JSON PARSING

```
//Builds the state of a game from the games JSON file
public class JSONMaster {
 /**
  * Write a game and its contents into a JSON file.
 * @param the game to be written and the file path to where you would like the JSON file to be
 * saves and the name of the JSON file
  */
        public void writeToJSON (Game g, String fileName)
```

```
    /**
     * Given a file path, read in a JSON file and construct a game with that data
     * @param filePath
     */
    public Game readFromJSON (String jsonFileLocation)
```

MOVEMENT

```
/**
 * Defines the movement of a piece. Responsible for maintaining the behavior
 * properties of a piece and executing allowed movements
 */
public class Movement {
    public Movement(List<Point2D>... endPoints)
    //Constructor taking in Point2Ds representing all possible relative locations of movement

    /**
     * Return absolute possible x,y coordinates of movement based on current
     * location x,y
     */
    public List<Point2D> getPossibleLocs(int x, int y)

/**
 * At every point on the piece's movement path, this calculates which direction the piece
 * should be facing. Also if you want to turn the piece to face an enemy or something like
 * that, you simply enter the location of the piece and the location of the enemy.
 */
public class Orientator

    /**
    * Calculates the amount that the piece needs to turn between each unit of movement
    * so that it is facing the proper direction
     */
    public void calculateTurn(double currentFacing, Point2D from, Point2D to)

    /**
     * Turns the piece
     */
    public void turn(Piece p)

    /**
     * Get the turn
     */
    public double getTurn()

/**
 * Defines a path(An arrangement of Points that must be traveresed during a
 * movement) for movement.
 */
public class Path
    /**
     * Default Constructor
     */
    public Path()

    /**
     * Constructor
     */
    public Path(List<Point2D> myPath)

    /**
     * Adds Point2Ds to path
     */
    public void addPointsToPath(Point2D... args)

    /**
     * Removes a point at a given index from the Path
     */
    public void removePointsFromPath(int index)

    /**
     * Returns a List of Point2Ds corresponding to the Path
     */
    public List<Point2D> getPath()
```

PATCH

```
public abstract class Patch
        /**
         * Constructor for patch
         * @param state of patch (this is more like type of patch: ex. fire, water, etc)
         * @param id of patch (each patch has its unique ID)
         * @param imageLocation of patch(form like "images/myImage.jpg")
         * @param p coordinate of patch
         */
        public Patch(int state, int id, String imageLocation, Point2D p);
```

```
/**
 * Getter for state
 * @return patch's state
 */
public int getMyState();

/**
 * Sets patch's state
 * @param myState of patch
 */
public void setMyState(int myState);

/**
 * Getter for ID
 * @return ID of patch
 */
public int getMyID();

/**
 * Getter for patch's coordinate location
 * @return coord of patch
 */
public Point2D getLoc();

/**
 * Sets patch's ID
 * @param myID of patch
 */
public void setMyID(int myID);

/**
 * Getter for patch's image
 * @return image of the patch
 */
public ImageView getImageView();

/**
 * Sets chosen image to patch's location
 * @param imageLocation image file's location
 */
public void setMyImage(String imageLocation);
```

PIECE

```
/**
 * Class for pieces. Pieces are the primary unit for game play.
 * They have movement and can carry out various actions during the game.
 */
public class Piece
	/**
	 * Piece constructor
	 *
	 * @param imageLocation - url of the piece's image location
	 * @param m - List of Movement defining how/where the
	 * piece moves relative to its current position
	 * @param a - List of Actions defining what actions are available
	 * for each piece to perform
	 * @param stats - the Piece's stats, already defined
	 * @param p - Point2D containing the piece's current coordinates
	 * @param tid - Piece's type ID, serves as a reference to this type of piece
	 * @param uid - Piece's unique ID, serves as a reference to this specific instance of piece
	 * @param pid - Piece's player ID, serves as a reference to which player
	 * this piece belongs to
	 * @param inventory - Piece's inventory if the user chooses to use an inventory
	 */
	public Piece(String imageLocation, List<Movement> m, List<Action> a, Stats stats,
			Point2D p, int tid, int uid, int pid, Inventory inventory);

	/**
	 * Returns the image location url (for data saving)
	 */
	public String getImageLocation();

	/**
	 * Returns the ImageView of the piece for display
	 */
	public ImageView getImageView();

	/**
	 * Returns the int ID for this type of piece
	 */
	public int getTypeID();

	/**
	 * Returns the int ID for this instance of piece
	 */
	public int getUniqueID();
```

```
/**
 * Returns the int ID for the player controlling this piece
 */
public int getPlayerID();

/**
 * Sets the piece's location to the specified Point2D
 * @param p - Point2D of the piece's new location
 */
public void setLoc(Point2D p);

/**
 * Returns the Point2D indicating the piece's coordinates
 */
public Point2D getLoc();

/**
 * Returns the piece's stats
 */
public Stats getStats();
/**
 * Adds an Action to the piece's list of Actions
 */
public void addAction(Action a);

/**
 * Removes an Action from the piece's list of Actions
 */
public void removeAction(Action a);

/**
 * Returns the list of the piece's available actions. Takes into account inventory if relevant.
 * @return List of available actions
 */
public List<Action> getActions();

/**
 * Marks the myShouldRemove boolean to true to flag for piece removal from board
 */
public void markForRemoval();

/**
```

         * Checks if the piece should be removed
         * @return boolean for whether or not the piece should be removed
         */
        public boolean **shouldRemove**();

        /**
         * Adds an item to the inventory as long as there is an inventory and the item added
         * is not the piece holding the inventory.
         * @param item - piece to be added to inventory
         * @return boolean stating whether item was added
         */
        public boolean **addToInventory**(Piece item);

        /**
         * Removes an item form the inventory as long as there is an inventory and the item added
         * is not the piece holding the inventory.
         * @param item - piece to be removed from the inventory
         */
        public void **removeFromInventory**(Piece item);

RULES

/**
 * A rule defines when a user's turn is over
 */
public class **Rule**
        public **boolean conditionsMet** (int x);
        //Checks to see if the Rule condition has been satisfied or not

STATS

/**
 * Numerical stats class. Stats are contained in every piece. Stats map a stat name to a double value.
 */
public class **Stats**
        /**
         * Stats constructor for initializing empty stats map
         */
        public **Stats**();

        /**
         * Stats constructor for initializing with already
         * created map of names to doubles
         * @param stats - map of stat names to doubles
         */

```
public Stats(Map<String, Double> stats);

/**
 * Adds a new stat to the stats map
 */
public void add(String name, double value);

/**
 * Removes a stat from the stats map by name
 */
public void remove(String name);

/**
 * Gets the value of the stat indicated by name
 */
public double getValue(String name);

/**
 * Sets the value of the stat with the indicated name to the value specified
 */
public void setValue(String name, double value);

/**
 * Clears the stats map
 */
public void clear();

/**
 * Returns a map of all the stats
 */
public Map<String, Double> getStatsMap();
```