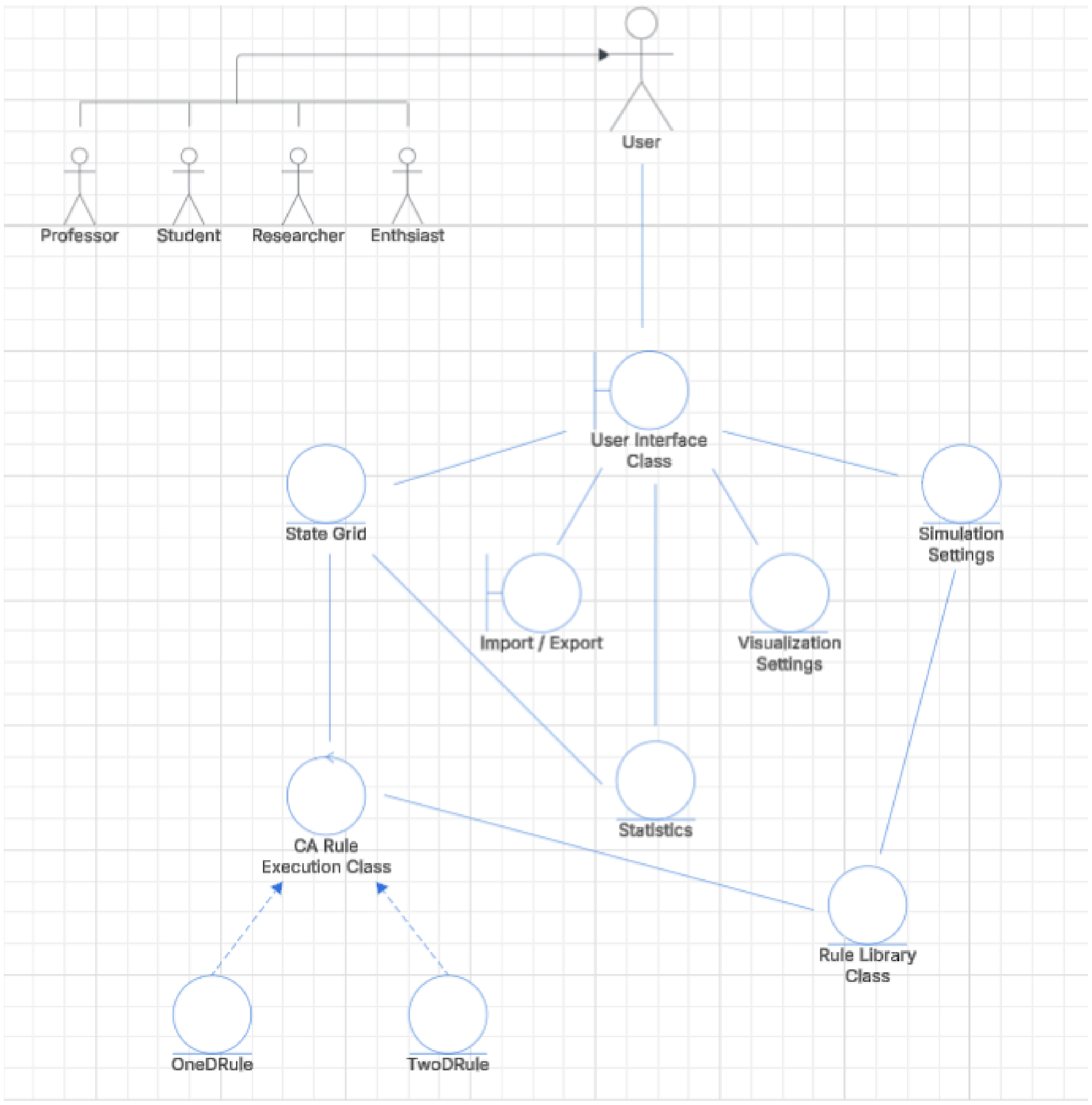
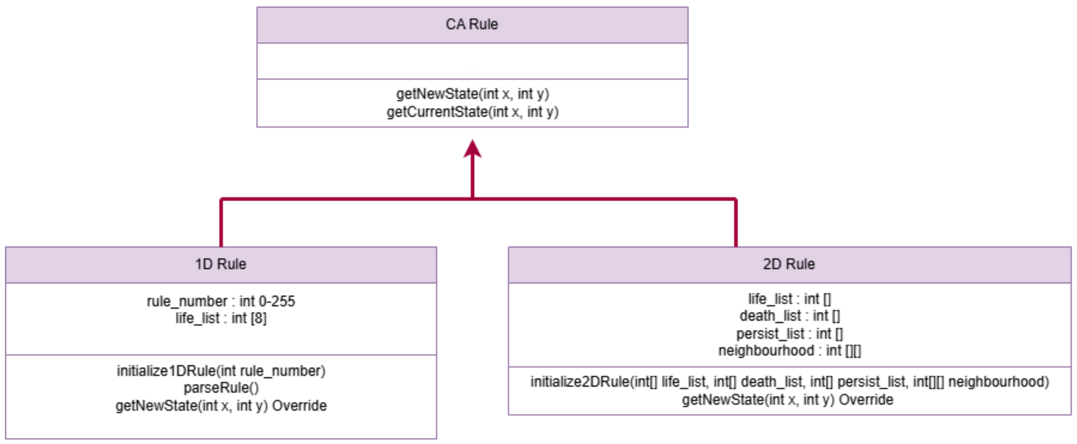
Finalization of Class Diagram

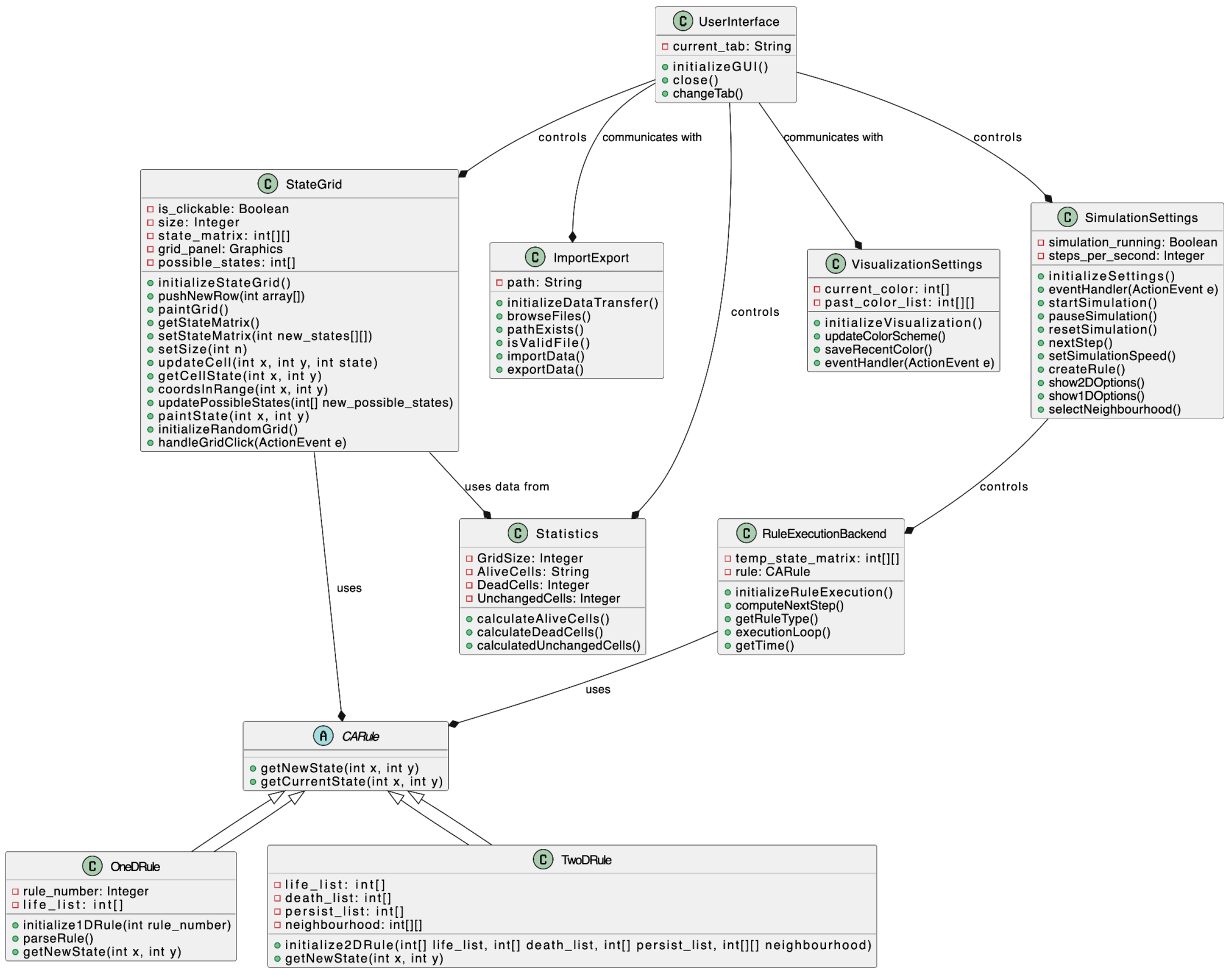
- **Overall class diagram** that shows the primary classes and their relationships with each other after the format of diagram 14.13. This should be mostly complete from the OOA but may require revisions or the addition of new classes.



- **Inheritance diagram:** Any parent classes that may have specific children classes should each have a diagram demonstrating how the children inherit from the parent and what overrides or modifications are made. Figure 14.14 for attributes and 14.15 for methods


CA Rule
getNewState(int x, int y)
getCurrentState(int x, int y)
1D Rule
rule_number : int 0-255
life_list : int [8]
initialize1DRule(int rule_number)
parseRule()
getNewState(int x, int y) Override
2D Rule
life_list : int []
death_list : int []
persist_list : int []
neighbourhood : int [][]
initialize2DRule(int[] life_list, int[] death_list, int[] persist_list, int[][] neighbourhood)
getNewState(int x, int y) Override

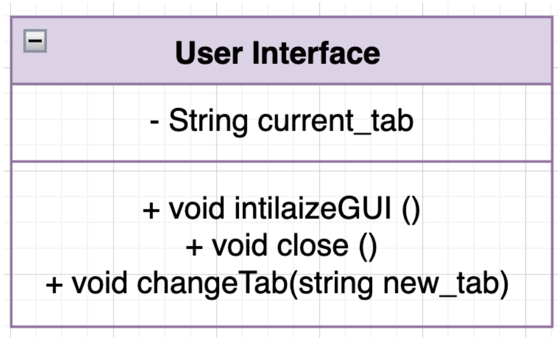
- **Detailed class diagram:** Diagram that includes each of the classes within the application, their internal methods, and the communication between them. It differs from the overall class diagram by providing more specifics about each class.



Final List of Classes

Class: User interface

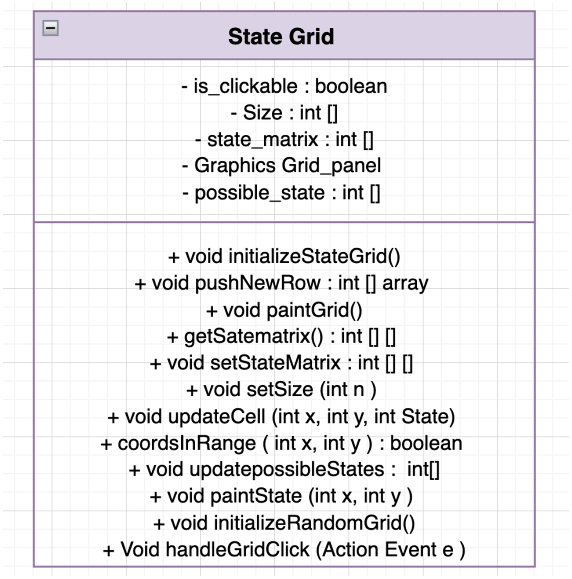
- **Purpose:** Boundary class that mediates ALL the user interactions with the application. It contains the eventhandler, which takes an interaction from the user and delegates the task to the appropriate class. The GUI will extend the JFrame class in Java Swing.
- **List of methods:**
    - initializeGUI()
        - Creates instances of all the subcomponents like the state grid, settings, and visualization tabs. And positions them in its JFrame.
    - close()
        - Halts any running operations and closes the gui
    - changeTab()
        - Switches focus to the new requested tab
    - Maximize() / Minimize()
        - Handled by Java swing
- **List of attributes**
    - current_tab (string): contains the id of the tab that is currently in focus.



Class: State grid

- **Purpose:** Place to display dynamic simulation results and obtain user click inputs for initial condition setting.
- **List of methods:**
    - initializeStateGrid()
        - Creates the state grid components and connects all the interactive widgets to event handlers
    - pushNewRow(int array[])
        - Take a new row of dimension (1 x size) and pushes a new row to the top of the state grid
    - paintGrid()
        - Reads from the state matrix and updates each cell in the grid_panel accordingly.

        - getStateMatrix()
            - Returns the state_matrix for data exporting
        - setStateMatrix(int new_states[][])
            - Takes a new properly sized matrix to replace its current matrix. The set state matrix also calls the update grid method.
        - setSize(int n)
            - If the simulation is not running, this creates a new state_matrix and with the proper size and destroys the old one. that when the size is changed, it preserves the states of the top left and grows/shrinks from the top right.
        - updateCell(int x, int y, int state)
            - If the new state is valid, this updates the cell at (x,y) to the new state value. Note that y is the row and x is the column in the matrix.
        - getCellState(int x, int y)
            - Returns the cell state at (x, y) that is (col, row)
        - coordsInRange(int x, int y)
            - Returns true if the (x, y) coordinates are in the matrix range.
        - updatePossibleStates(int[] new_possible_states)
            - Changes the possible states to the new possible states list.
        - paintState(int x, int y)
            - Paints the state in the grid panel to match the state in the state_matrix
        - initializeRandomGrid()
            - Assigns each cell random values.
        - handleGridClick(ActionEvent e)
            - Takes the click event and determines which cell was clicked using the mouse coordinates. If the simulation is not running it will update the cell state from the current to the next cell state in the possible_states list.
    - **List of attributes**
        - is_clickable (boolean): false if the simulation is running, true if it is paused
        - size (int): side length of cell grid. Please note that the grid is always square.
        - state_matrix (int[][]): array of order (size x size) each containing an integer representing that cell's state.
        - grid_panel (Graphics g): The graphical grid that gets repainted based on the states
        - possible_states (int[]): List of valid states. For the moment we will just have 2, which are 0 and 1, but in the future it could be more.
    - **Notes:**
        - The cells will not each have explicit classes, but rather will be represented as integer values in the array. The paintState will update the color of the corresponding cell in the gui.


State Grid
- is_clickable : boolean
- Size : int []
- state_matrix : int []
- Graphics Grid_panel
- possible_state : int []
+ void initializeStateGrid()
+ void pushNewRow : int [] array
+ void paintGrid()
+ getSatematrix() : int [] []
+ void setStateMatrix : int [] []
+ void setSize (int n)
+ void updateCell (int x, int y, int State)
+ coordsInRange (int x, int y) : boolean
+ void updatepossibleStates : int[]
+ void paintState (int x, int y)
+ void initializeRandomGrid()
+ Void handleGridClick (Action Event e)

Abstract Class: CA Rule

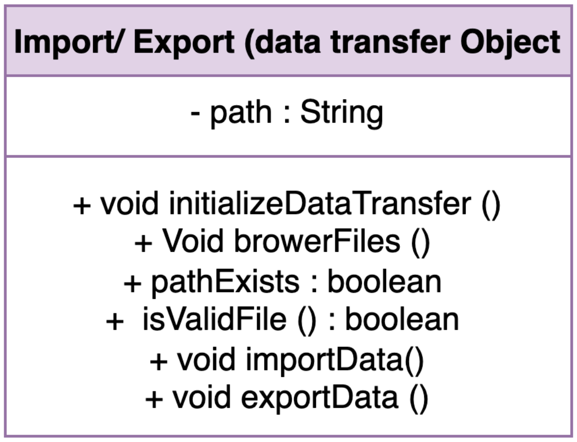
- **Purpose:** A rule contains a function that takes a cell's (x, y) that is (col, row) coordinates and returns the new state for that cell.
- **List of methods:**
    - getNewState(int x, int y)
        - If the coordinates are valid, this gets the new state for the cell at (x,y) based on its neighbourhood.
    - getCurrentState(int x, int y)
        - Gets the state of the cell at (x,y) with the additional functionality of wrapping around back to the first index using modular arithmetic if either x or y is out of range. This gives the rule execution the illusion of circularity.
- **1D rule (Child)**
    - **Purpose:** Evaluates a cell at one of the 256 Wolfram 1D CA rules. The neighborhood size of a 1D rule is just the cell itself and the 2 adjacent cells.
    - **List of methods**
        - initialize1DRule(int rule_number)
            - Sets the rule number and parses the active_list based on the binary representation of the rule number.
        - parseRule()
            - Goes through the 8 bits 0 - 7 of the rule number and if the nth bit is 1 add that to the list.
        - getNewState(int x, int y) override
            - Creates a binary number from the states of the neighbourhood with state at (x-1,y) as the least significant digit and (x+1,y) as the most significant. If that number is in the active list return 1 otherwise 0.
    - **List of attributes**

        - rule_number (int 0-255): wolfram rule number
        - life_list (int[]): This is the list of numbers that give an active (alive) 1 state. For the Wolfram CA this is a set with numbers from 0-7 determined by where the “1s” are in the binary representation of the rule number.
- **2D rule (Child)**
    - **Purpose:** Evaluates a cell at a more sophisticated 2D CA rule such as the game of life. To begin with we will only do number of dead / alive neighbor based rules since this will be the most simple to implement.
    - **List of methods:**
        - initialize2DRule(int[] life_list, int[] death_list, int[] persist_list, int[][] neighbourhood)
            - Sets each of the lists in the class.
        - getNewState(int x, int y) override
            - Counts the number of alive cells in the neighbourhood. With this count and the cell of interest state we use our lists to determine what the next cell state should be.
    - **List of attributes:**
        - life_list (int[]) number of alive neighbours required to turn a cell from dead to alive.
        - death_list (int[]): number of alive neighbours which will result in death of the cell.
        - persist_list (int[]): number of alive neighbours required to sustain life. Please note that this may not necessarily bring a dead cell to life.
        - neighbourhood (int[][]): list of 2D coordinate shifts that define a neighbor's position relative to the cell of interest. For example, Conway's game of life has a neighbourhood of {[-1,0], [1,0], [0,-1], [0,1]}.
- **Notes:**
    - These classes at the moment are only designed for 2-state binary cells and would require some revision to move to a higher number of states or a higher state class would need to be created.

Class: Import / Export (Data Transfer Object)

- **Purpose:** Allow the user to import / export rules and initial states into the application.
- **List of methods**
    - initializeDataTransfer()
        - Creates all the widgets and buttons that will be used for the import and exporting. These include an import and export button along with a file path textbox and file browsing button.
    - browseFiles()
        - Opens a file browsing dialogue window and allows the user to select their desired destination. Once selected this sets the data path.
    - pathExists()
        - Returns true if the file path exists otherwise false.

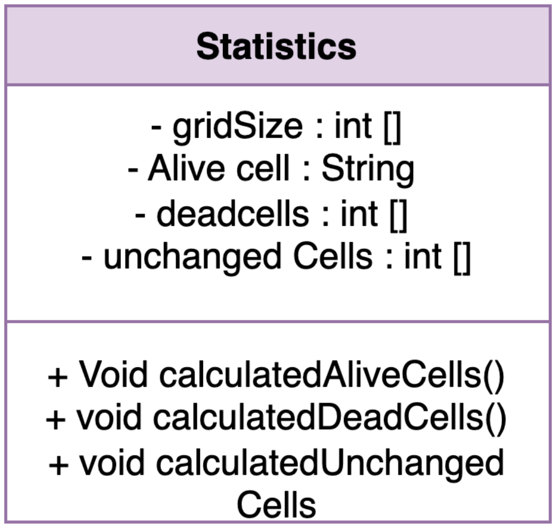
        - isValidFile()
            - Checks if the file path exists, is in the xml format, and has entries of initial conditions, simulation settings, and the rule.
        - importData()
            - If the file is valid this first loads the settings like rule dimension, grid size etc. Then it loads the initial conditions into the state grid and finally it loads the rule into the rule execution class.
        - exportData()
            - Save the stategrid, current rule and settings into an xml file and saves it at the path. If the path is invalid a the user is met with an error popup and if it already exists a warning popup is created and waits for the user to confirm their choice before overriding.
    - **List of attributes**
        - path (string): absolute file path for importing or exporting



Class: Statistics

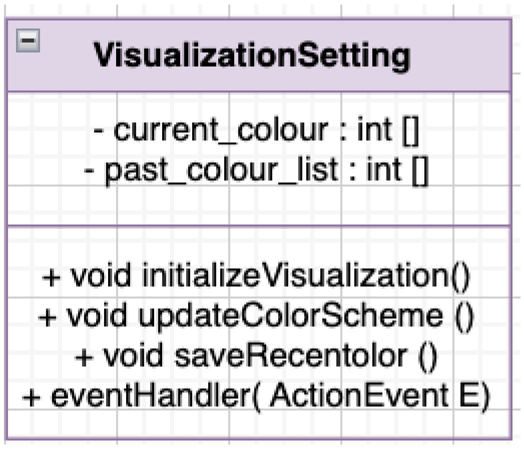
- **Purpose:** Provide the user with potentially interesting information about the current simulation
- **List of methods:**
    - calculateAliveCells()
    - calculateDeadCells()
    - calculatedUnchangedCells()
- **List of attributes:**
    - GridSize (int): attribute holding grid size value (n x n)
        - Note: Gets this value from simulation settings class function when user inputs the grid size (n) they want and its processed
    - AliveCells (string): string holding the percentage of alive cells as “%number”
    - DeadCells (int): int number holding how many cells from the previous generation have died in the new generation (1 to 0 or have gone from black to white)

        - UnchangedCells (int): int number holding how many cells from the previous generation have stayed the same in the new generation (1 to 1, 0 to 0 or black to black, white to white)


Statistics
- gridSize : int []
- Alive cell : String
- deadcells : int []
- unchanged Cells : int []
+ Void calculatedAliveCells()
+ void calculatedDeadCells()
+ void calculatedUnchanged Cells

Class: Visualization Settings

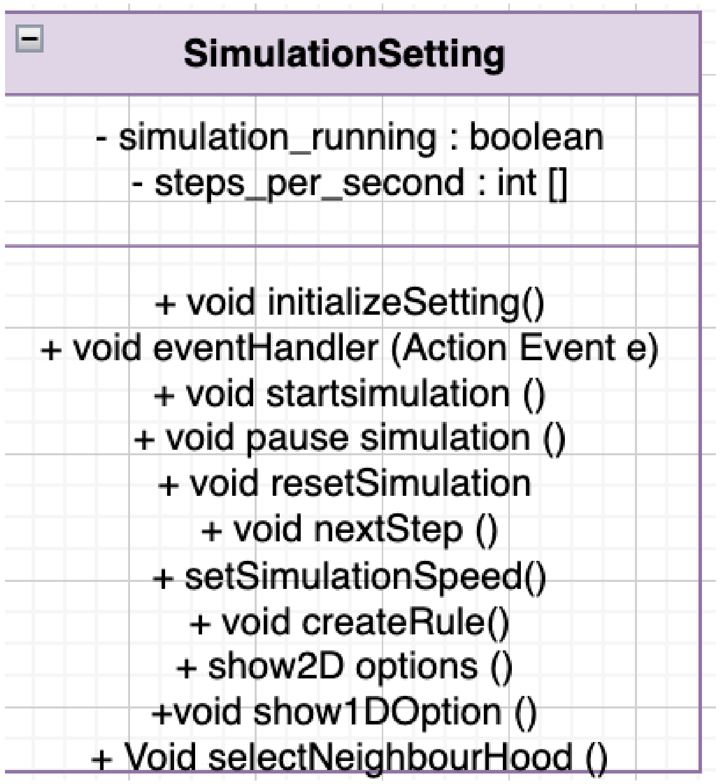
- **Purpose:** Provide an interface and functionality for the user to customize the application aesthetics.
- **List of methods:**
    - initializeVisualization()
        - Creates the visual color grid and other elements of aesthetics customization.
    - updateColorScheme()
    - saveRecentColor()
    - eventHandler(ActionEvent e)
- **List of attributes:**
    - current_color (int[]) array of 3 values corresponding to r, g, b
    - past_color_list (int[][]) array containing recently selected rgb values.


VisualizationSetting
- current_colour : int []
- past_colour_list : int []
+ void initializeVisualization()
+ void updateColorScheme ()
+ void saveRecentolor ()
+ eventHandler(ActionEvent E)

Class: Simulation Settings

- **Purpose:** Gets and stores the options set by the user from the settings tab in the GUI and then reacts appropriately when changes are made to the settings.
- **List of methods:**
    - initializeSettings()
    - eventHandler(ActionEvent e)
    - startSimulaton()
        - Sets the running simulation flag to true
    - pauseSimulation()
        - Sets the running flag to false
    - resetSimulation()
        - Clears the rule, the step counter, and sets the state grid back to zero values.
    - nextStep()
        - Calls the next step in the backend
    - setSimulationSpeed()
        - Reads the setting from the simulation speed slider and updates the steps per second.
    - createRule()
        - Creates an instance of the 1D or 2D class based on the 1D/2D setting in the gui.
    - show2DOptions()
        - Displays entry boxes that allows the user to enter values for the number of neighbours for life, death, and persistence in the 2D rule. Also displays the select neighbourhood button.
    - show1DOptions()
        - Displays a numerical entry box that takes values from 0 - 255 for the 1D rule.
    - selectNeighbourhood()

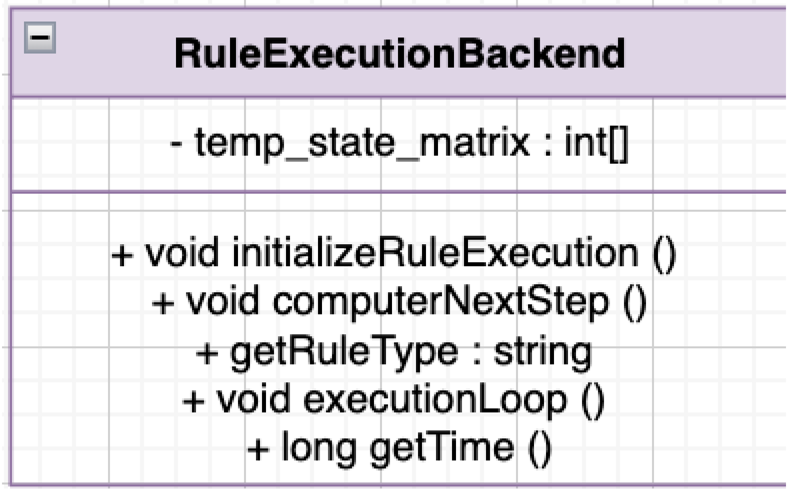
            - Displays the neighbourhood selection screen which is a 9x9 grid where the central cell would be the cell of interest and any other cell that the user selects in the 9x9 grid is part of the neighbourhood.
- **List of attributes:**
    - simulation_running (bool) flag for whether or not the simulation is started or not.
    - steps_per_second (int)


SimulationSetting
- simulation_running : boolean
- steps_per_second : int []
+ void initializeSetting()
+ void eventHandler (Action Event e)
+ void startsimulation ()
+ void pause simulation ()
+ void resetSimulation
+ void nextStep ()
+ setSimulationSpeed()
+ void createRule()
+ show2D options ()
+void show1DOption ()
+ Void selectNeighbourHood ()

Class: Rule Execution Backend

- **Purpose:** Contains the timers and loop that will run the simulation and compute the next gird state based on the rule and the current state grid.
- **List of methods:**
    - initializeRuleExecution()
        - Allocates memory for the temp_state_matrix
    - computeNextStep()
        - Iterates through each cell in the current state and computes the next state for each cell by calling the rule. Once the execution is done we either write the next row for 1D or the entire temp_matrix to the state grid for 2D.
    - getRuleType()
        - Determines whether the rule is 1D or 2D so that we can perform condition execution accordingly.
    - executionLoop()

        - Runs an infinite loop with a timer that checks whether simulation is started and if the time since the last rule execution exceeds the time between executions.
    - getTime()
        - Returns the current time to the resolution of milliseconds.
- **List of attributes:**
    - temp_state_matrix (int[][]) auxiliary state matrix that the backend will work with and then write to the state_matrix in the state grid once all the computations in the step are complete.
    - rule (CARule class): either 1D or 2D rule class set by the user.


RuleExecutionBackend
- temp_state_matrix : int[]
+ void initializeRuleExecution ()
+ void computerNextStep ()
+ getRuleType : string
+ void executionLoop ()
+ long getTime ()

Detailed Class Design

Detailed Design for the UserInterface class

```
public void UserInterfaceEvent( )
{
   UserInterface::initializeGUI;

   userChoice = EventHandler.getUserChoice( );

   if (userChoice is changeTab)
   {
      newTab = userChoice.TabID
      UserInterface::changeTab(newTab);
      UserInterface::current_tab = newTab;
   }
   else if (userChoice is close)
   {
      UserInterface::close;
   }
   else if (userChoice is minimize) {
      UserInterface::minimize;
   }
   else if (userChoice is maximize) {
      UserInterface::maximize;
   }
   else
   {
      System.out.println("Invalid choice. Please try again.");
   }
}
```

Detailed Design for the CARule Class

```
public void CARuleEventHandler()
{
    userChoice = EventHandler.getUserChoice();

    if (userChoice is initialize1DRule)
    {
        ruleNumber = EventHandler.getUserInput();
        CARule1D::initialize1DRule(ruleNumber);
        CARule1D::parseRule;
        System.out.println("1D Rule #" + ruleNumber + " initialized.");
    }
    else if (userChoice is initialize2DRule)
    {
        lifeList = EventHandler.getUserInput();
        deathList = EventHandler.getUserInput();
        persistList = EventHandler.getUserInput();
        neighbourhood = EventHandler.getUserInput();
        CARule2D::initialize2DRule(lifeList, deathList, persistList, neighbourhood);
        System.out.println("2D Rule initialized.");
    }
    else if (userChoice is getNewState)
    {
        x = EventHandler.getUserInput();
        y = EventHandler.getUserInput();
        newState = CARule::getNewState(x, y);
        System.out.println("New state at (" + x + "," + y + "): " + newState);
    }
    else if (userChoice is getCurrentState)
    {
        x = EventHandler.getUserInput();
        y = EventHandler.getUserInput();
        currState = CARule::getCurrentState(x, y);
        System.out.println("Current state at (" + x + "," + y + "): " + currState);
    }
    else
    {
        System.out.println("Invalid rule option. Please try again.");
    }
}
```

Detailed Design for the Import/Export class

```
public void ImportExportEventHandler()
{
    ImportExport::initializeDataTransfer;

    userChoice = EventHandler.getUserChoice();

    if (userChoice is browseFiles)
    {
        ImportExport::browseFiles;
        userPathChoice = EventHandler.getUserChoice();

        if (ImportExport::pathExists(userPathChoice) is TRUE)
        {
                    ImportExport::path = userPathChoice;
        }
        else
        {
            System.out.println("Selected path does not exist.");
        }
    }
    else if (userChoice is importFile)
    {
        userImportChoice = EventHandler.getUserChoice();
        if (ImportExport::isValidFile(userImportChoice) is TRUE)
        {
            ImportExport::importData(userImportChoice);
        }
        else
        {
            System.out.println("Invalid file. Import failed.");
        }
    }
    else if (userChoice is exportFile)
    {
        if (ImportExport::pathExists is FALSE)
        {
            System.out.println("Invalid export path. Please browse and select a valid destination.");
        }
        else
        {
            if (FileManager.fileAlreadyExists(ImportExport.path) is TRUE)
            {
                UserPrompt::showWarning("File already exists. Overwrite?");
                if (UserPrompt::userConfirmed( ) is TRUE)
                {
                    ImportExport::exportData(ImportExport.path);
                }
                else
                {
                    System.out.println("Export canceled by user.");
                }
            }
            else
            {
                ImportExport::exportData(ImportExport.path);
            }
        }
    }
    else
    {
        System.out.println("Invalid choice. Please try again.");
    }
```

Detail Design for StateGrid Class

```
public void StateGridEventHandler()
{
    StateGrid::initializeStateGrid;

    userChoice = EventHandler.getUserChoice();
```

```
if (userChoice is pushNewRow)
{
   newRow = EventHandler.getUserInput();
   StateGrid::pushNewRow(newRow);
   StateGrid::paintGrid;
}
else if (userChoice is paintGrid)
{
   StateGrid::paintGrid;
}
else if (userChoice is getStateMatrix)
{
   matrix = StateGrid::getStateMatrix();
   System.out.println("State Matrix Retrieved.");
}
else if (userChoice is setStateMatrix)
{
   newMatrix = EventHandler.getUserInput();
   StateGrid::setStateMatrix(newMatrix);
   StateGrid::paintGrid;
}
else if (userChoice is setSize)
{
   newSize = EventHandler.getUserInput();
   StateGrid::setSize(newSize);
   StateGrid::paintGrid;
}
else if (userChoice is updateCell)
{
   x = EventHandler.getUserInput();
   y = EventHandler.getUserInput();
   newState = EventHandler.getUserInput();
   StateGrid::updateCell(x, y, newState);
   StateGrid::paintState(x, y);
}
else if (userChoice is getCellState)
{
   x = EventHandler.getUserInput();
   y = EventHandler.getUserInput();
   cellState = StateGrid::getCellState(x, y);
   System.out.println("Cell state at (" + x + "," + y + "): " + cellState);
}
else if (userChoice is coordsInRange)
{
        x = EventHandler.getUserInput();
        y = EventHandler.getUserInput();
        isInRange = StateGrid::coordsInRange(x, y);
        System.out.println("Coordinates (" + x + "," + y + ") in range: " + isInRange);
    }
    else if (userChoice is updatePossibleStates)
    {
        newStates = EventHandler.getUserInput();
        StateGrid::updatePossibleStates(newStates);
    }
    else if (userChoice is paintState)
    {
        x = EventHandler.getUserInput();
        y = EventHandler.getUserInput();
        StateGrid::paintState(x, y);
    }
    else if (userChoice is initializeRandomGrid)
    {
        StateGrid::initializeRandomGrid();
        StateGrid::paintGrid;
    }
    else if (userChoice is handleGridClick)
    {
        clickEvent = EventHandler.getUserClickEvent();
        StateGrid::handleGridClick(clickEvent);
        StateGrid::paintGrid;
    }
    else {
        System.out.println("Invalid Selection")
    }
}
```

Detailed Design for the VisualizationSettings class

```
public void VisualizationSettingsEvent()
{
    VisualizationSettings::initializeVisualization;

    userChoice = EventHandler.getUserChoice();

    if (userChoice is updateColourScheme)
    {
        newColor = EventHandler.getUserInput();
        if (newColor is valid color)
        {
            VisualizationSettings::updateColorScheme(newColor);
            VisualizationSettings::saveRecentColor(newColor);
        }
        else
        {
            System.out.println("Invalid color selection. Please choose a valid RGB value.");
        }
    }
    else if (userChoice is saveRecentColor)
    {
        lastColor = VisualizationSettings::current_color;
        VisualizationSettings::saveRecentColor(lastColor);
    }
    else if (userChoice is selectPastColor)
    {
        if (VisualizationSettings::past_color_list is not empty)
        {
            selectedColor = EventHandler.getUserInput();

            if (selectedColor is valid color)
            {
                VisualizationSettings::updateColorScheme(selectedColor);
            }
            else
            {
                System.out.println("Invalid selection. Please try again.");
            }
        }
        else
        {
            System.out.println("No past color schemes available.");
        }
    }
    else
    {
        System.out.println("Invalid choice. Please try again.");
    }
}
```

Detailed Design for the SimulationSettings Class

```
public void SimulationSettingsEventHandler()
{
   SimulationSettings::initializeSettings;

   userChoice = EventHandler.getUserChoice();

   if (userChoice is startSimulation)
   {
      SimulationSettings::startSimulaton;
      System.out.println("Simulation started.");
   }
   else if (userChoice is pauseSimulation)
   {
      SimulationSettings::pauseSimulation;
      System.out.println("Simulation paused.");
   }
   else if (userChoice is resetSimulation)
   {
      SimulationSettings::resetSimulation;
      StateGrid::paintGrid;
      System.out.println("Simulation reset.");
   }
   else if (userChoice is nextStep)
   {
      SimulationSettings::nextStep;
      StateGrid::paintGrid;
   }
   else if (userChoice is setSimulationSpeed)
   {
      newSpeed = EventHandler.getUserInput();
      SimulationSettings::setSimulationSpeed(newSpeed);
      System.out.println("Speed updated to " + newSpeed + " steps/sec.");
   }
   else if (userChoice is createRule)
   {
      ruleType = EventHandler.getUserChoice();
      SimulationSettings::createRule(ruleType);
      System.out.println(ruleType + " rule created.");
   }
   else if (userChoice is show2DOptions)
   {
      SimulationSettings::show2DOptions;
   }
   else if (userChoice is show1DOptions)
```

```
    {
        SimulationSettings::show1DOptions;
    }
    else if (userChoice is selectNeighbourhood)
    {
        SimulationSettings::selectNeighbourhood;
    }
    else
    {
        System.out.println("Invalid simulation setting. Please try again.");
    }
}
```

Detailed Design for the RuleExecutionBackend Class

```
public void RuleExecutionEventHandler()
{
    RuleExecutionBackend::initializeRuleExecution;

    userChoice = EventHandler.getUserChoice();

    if (userChoice is startSimulation)
    {
        if (RuleExecutionBackend::simulationRunning is FALSE)
        {
            RuleExecutionBackend::startSimulation();
            System.out.println("Simulation started.");
        }
        else
        {
            System.out.println("Simulation is already running.");
        }
    }
    else if (userChoice is stopSimulation)
    {
        if (RuleExecutionBackend::simulationRunning is TRUE)
        {
            RuleExecutionBackend::stopSimulation();
            System.out.println("Simulation stopped.");
        }
        else
        {
            System.out.println("Simulation is not currently running.");
        }
    }
    else if (userChoice is stepForward)
    {
        if (RuleExecutionBackend::simulationRunning is FALSE)
        {
            RuleExecutionBackend::computeNextStep();
            Visualization::updateGridDisplay();
            System.out.println("Simulation stepped forward.");
        }
        else
        {
            System.out.println("Cannot step forward while simulation is running.");
        }
    }
    else if (userChoice is setSpeed)
    {
        newSpeed = EventHandler.getUserInput();
        RuleExecutionBackend::setExecutionSpeed(newSpeed);
        System.out.println("Simulation speed updated to " + newSpeed + " ms.");
    }
    else
    {
        System.out.println("Invalid choice. Please try again.");
    }
}
```

Detailed Design for the Statistics Class

```
public void StatisticsEventHandler()
{
    Statistics::initializeStatistics;

    userChoice = EventHandler.getUserChoice();

    if (userChoice is computeCellCounts)
    {
        aliveCells = Statistics::getAliveCellCount();
        deadCells = Statistics::getDeadCellCount();
        totalCells = aliveCells + deadCells;
        System.out.println("Alive Cells: " + aliveCells + ", Dead Cells: " + deadCells + ", Total: " +
totalCells);
```

```
    }
    else if (userChoice is computePopulationDensity)
    {
        density = Statistics::calculatePopulationDensity();
        System.out.println("Current Population Density: " + density + "%");
    }
    else if (userChoice is computeEntropy)
    {
        entropy = Statistics::calculateEntropy();
        System.out.println("Grid Entropy: " + entropy);
    }
    else if (userChoice is trackStateChanges)
    {
        generation = EventHandler.getUserInput();
        changes = Statistics::getStateChanges(generation);
        System.out.println("State changes in generation " + generation + ": " + changes);
    }
    else if (userChoice is getSimulationHistory)
    {
        history = Statistics::retrieveSimulationHistory();
        System.out.println("Simulation History Retrieved.");
    }
    else if (userChoice is exportStatistics)
    {
        if (ImportExport::pathExists is FALSE)
        {
            System.out.println("Invalid export path. Please select a valid destination.");
        }
        else
        {
            ImportExport::exportData(Statistics::generateStatisticsReport());
            System.out.println("Statistics exported successfully.");
        }
    }
    else
    {
        System.out.println("Invalid choice. Please try again.");
    }
}
```