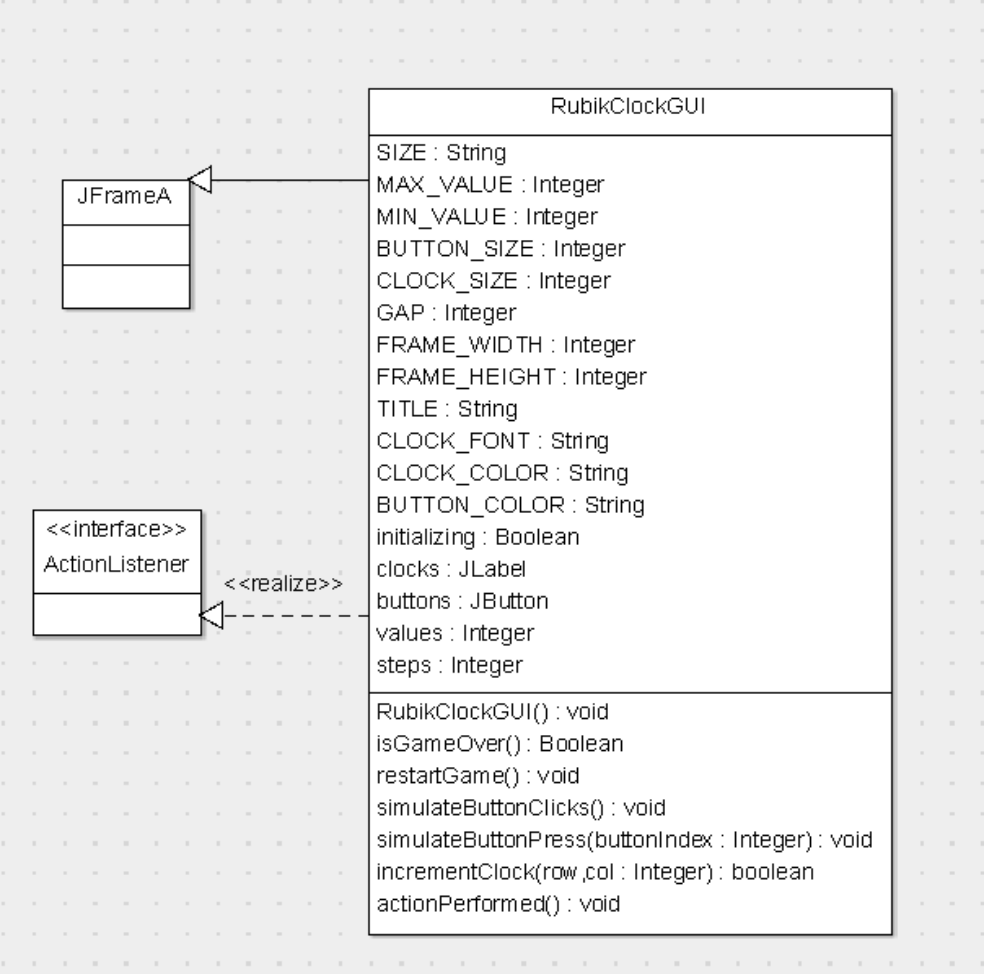
Isroilbek Jamolov **2. assignment/5. task** 15th Nobember 2023
DXFV5Y

dxfv5y@inf.elte.hu
Group 1

Task

Create a game, which implements the Rubik clock. In this game there are 9 clocks. Each clock can show a time between 1 and 12 (hour only). Clocks are placed in a 3x3 grid, and initially they set randomly. Each four clocks on a corner has a button placed between them, so we have four buttons in total. Pressing a button increase the hour on the four adjacent clocks by one. The player wins, if all the clocks show 12. Implement the game, and let the player restart it. The game should recognize if it is ended, and it has to show in a message box how much steps did it take to solve the game. After this, a new game should be started automatically.

UML diagram



Description of class methods

- RubicClockGUI (Constructor): Initializes the GUI for the Rubic Clock game. Sets up the frame, grid of clocks, and control buttons. All cloks are set to 12 initially, and then certain number of random button clicks (in my case it is 15, but it can modified easily) are simulated to start the game.
- isGameOver(): check if all clocks are set to 12, indicating the game has been solved. Returns 'true' if the game is over, otherweise 'false'.
- restartGame(): Resets the game to its initial state. All clocks are set to 12, and certain number of random button clicks (in my case it is 15) are simulated again. The 'steps' counter is reset to zero, and the 'initializing' flag is set for the duration of this process.
- simulateButtonClicks(): Simulates certain number of (in my case 15) random button clicks. This method is used durign the initialization and restart of the game to create a new puzzle configuration.
- simulateButtonPress(int buttonIndex): Simulates the pressing of a button, identified by 'buttonIndex'. It increments the value of the clocks affected by the button. If the game is not in the initializing phase, it incremetns the 'steps' counter.
- incrementClock(int row, int col): Increments the value of the clock at the specified 'row' and 'col'. If the clock's are value reaches the 'MAX_VALUE' (12), It is reset to 'MIN_VALUE' (1).
- actionPerformed(ActionEvent e): Overrides the 'ActionListener' interface's method to handle button click events. It identifies which button was clicked and calls 'simulateButtonPress'. It also check if the game is over after each button press.

These methods collectively manage the game's state, user interactions, and track the progress towards solving the puzzle. The event-driven nature of the game is primarily handled through 'actionPerformed', with 'simulateButtonPress' and 'incrementClock' updating reponse to user inputs.

Description of the connections between the event and event handlers

- Button Click Events: Each button (representing a corner of the Rubik Clock) is associated with a click event. This is achieved by adding the 'RubicClockGUI' class itself as an 'ActionListener' to each button during initialization in the constructor.
- Handling Button Clicks: The 'aactionPerformed(ActionEven e)' method is the event handler for button click events. When a button is clicked, this method is triggered. Inside 'actionPerformed', the method determines which button was clicked by comparing the source of the event ('e.getSource()') to each button in the 'buttons'' array.
- Simulating Button Press Logic: After identifying the clicked button, 'actionPerformed' calls 'simulateButtonPress(int buttonIndex)', passing the index of the clicked button. It calls 'incrementClock(int row, int col)' for each of the four clocks affected by the button.
- Updating Clock Values: The 'incremetnClock' method updates the value of a specific clock in response to a button press. If a clock's value reaches the maximum (12), it wraps around to the minimum (1).
- Checking Game State: At the end of the 'actionPerformed' method, there's a check to determine if the game is over (all clocks showing 12) by calling 'isGameOver()'. If the game is over, it displays a message to the player and calls 'restartGame()' to reset the game state.
- Restarting the Game: The 'restartGame' method resets all clocks to 12 and simulates certain number of random button pressess (15 in my case) to create a new starting state for the game. It also resets the step counter to zero.

This event-driven architecture allows for a responsive game interface where user interactions (button clicks) directly influence th egame state and lead to immediate visual feedback on the GUI.

Testcases

1. **Initial State Test**: Verify that at the start of the game, all clocks are not set to 12 due to the initial random button clicks. This checks the effetiveness of the 'simulateButtonClicks' method.
2. **Button Cluck Test**: Test each button to ensure that it correcly increments the values of its corresponding clocks. This validates the 'actionPerformed' and 'simulateButtonPress' methods.
3. **Clock Increment Test**: Verify that each clock correctly cycles from 1 to 12. And when it reaches the 12 then it should go back to 1.
4. **Winning Condition Test**: The game should end when all the clocks are 12 (so it shuold not overlap to 1 again)
5. **Count of Presses Testing**: Validate that at the end of the game the count is number of buttons and not the number of changes and resets to 0 when the game is restarted.
6. **Solvability Test**: If the number of randomly pressed numbers works with small numbers like 1 or 2 and so on, that means it works with bigger numbers as well.
   To check for small numbers we should just calculate the numbers, like B1 is randomly pressed then lables around B1 shuold be all 1 and the rest are 12.
7. **Solvability algorithm**: If the two elements of the beginning and the end of the row or columns' sum equal to middle one and the middle number shuold be odd or even depending on the number of random presses, and in our case as it 15, so the middle number of the 3x3 grid label shuold be odd number 3 (15%12 =3)