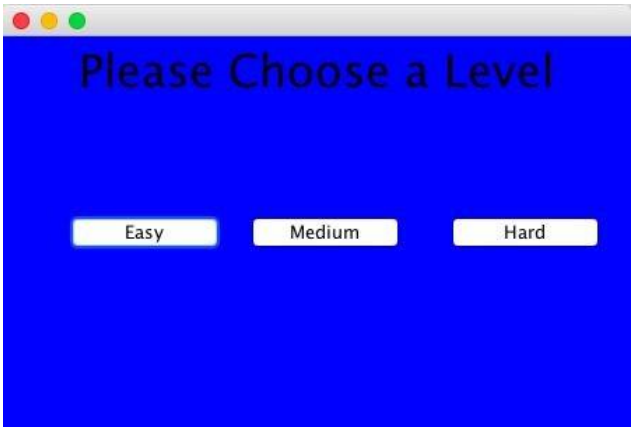
Development

Techniques Used:

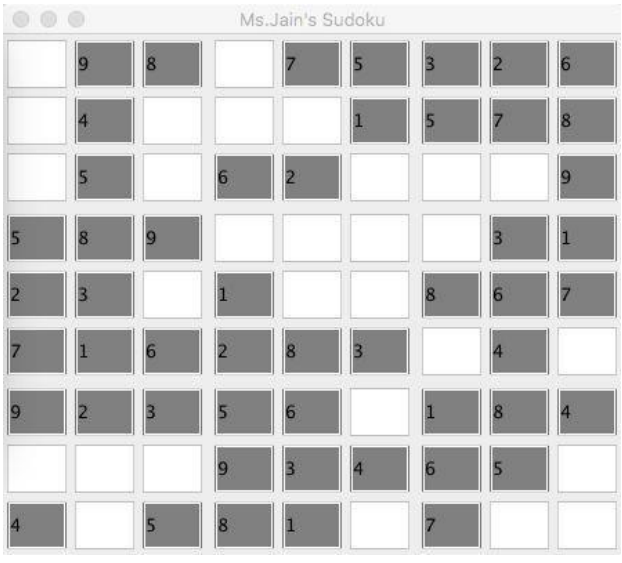
- Graphical User Interface (GUI)
- Algorithmic Thinking
- Recursion
- Object Oriented Programming (OOP)
- Existing Java Libraries
- Writing and Reading from Text Files.

Graphical User Interface (GUI) (and some Algorithmic Thinking)

I built the GUI for my program using the Java WindowBuilder extension on Eclipse. Using this extension, I was able to create JButtons, JLabels and JTextfields to guide the user through my program and allow him or her to interact with it.



When creating the main sudoku window, I wanted the already solved cells to have a grey background, and that they would not be editable by the player. As I iterated through a 2D array of JTexfields, and populated them with values, I set the background of the Textfield as white if there was no value (indicated by 0) and made it editable, while I set the background to gray if



there was a value and made that Textfield uneditable. This can be seen in the code below:

```
for(int r=0; r<9; r++) //copies the gamearray into the 2d array of Jtextfields.
{
    for(int c=0; c<9; c++)
    {
        cell[r][c]=new JTextField();
        if (LevelsWindow.gameArray[r][c] == 0)
        {
        cell[r][c].setText("");
        cell[r][c].setEditable( true ); //makes empty Jtextfields editable.
        }

        else
        {
        cell[r][c].setText(String.valueOf(LevelsWindow.gameArray[r][c])); //takes the string value of the index as
                                                                          //only strings can be stored in textfields.
        cell[r][c].setEditable( false );    //makes already solved Jtextfields uneditable.
        cell[r][c].setBackground(Color.gray);
        }
    }
}
```

I also created the Sudoku-like grid layout using a 3x3 2D array of 3x3 JPanels and I populated each JPanel with the nine JTextFields appropriate for that box. This can be seen here:

```
for(int x=0; x<3; x++) //iterates through the 2d array of type Jpanel.
{
    for(int y=0; y<3; y++)
    {
        for(int r=0; r<3; r++ ) //iterates through the internal 3x3 panels in each index of the panel array.
        {
            for(int c=0; c<3; c++)
            {
                panel[x][y].add(cell[r+x*3][c+y*3]); //adds textfields to each panel.
                                                     //In total 9 textfields added to each internal panel.
            }
        }
    add(panel[x][y]); //adds the current panel to the GridLayout.
    }
}
```

Algorithmic Thinking and Recursion

The main piece of algorithmic thinking in my program is in the SudokuGenerator class, which generates a completely random, new and valid (only one number from 1-9 in each row, column and 3x3 box) Sudoku with each game. The method that generates the sudoku is called MakeSudoku, it returns a 2D array and can be seen below:

```
public static int[][] MakeSudoku(int cellnumber, int[][] SudokuArray ) {
    if (cellnumber > 80)        // there are 81 cells in the sudoku board (0-80). so if the counter, 'cellnumber' exceeds 80,
       {return SudokuArray;}    // we have reached the last cell and the array can be returned.

    List<Integer> ANs = new ArrayList<Integer>(); //an arraylist of numbers which stores the possible values for each cell.

    for (int x = 1; x <10; x++)     // populating the arraylist with numbers 1 through 9.
        {
        ANs.add(x);
        }

    Collections.shuffle(ANs);   //shuffling the arraylist so that the selected value for each cell is random.

    int row = cellnumber / 9;   // finding the row and column values from the cellnumber.
    int col = cellnumber % 9;   // this is cited in Criterion C

        while (0<ANs.size())
        {
        int value = ValidValue(SudokuArray, col, row, ANs); //getting the random value for the current cell using the ValidValue method below.
        if (value == -1)                                    //if the ValidValue method return -1,
        {
            return null;                                    //null is returned and no value is stored in this cell.
        }
        else
        {
        SudokuArray[row][col] = value;                      // otherwise a valid value is returned and stored in that cell.
        }

        int[][] newSudokuArray = MakeSudoku(cellnumber + 1, SudokuArray);   //the value for the next cell is received by
                                                                            //recursively calling the same method with an increase in the cell number.
                                                                            //this value is stored in a 2D array simply since the method can only return
                                                                            //a 2D array type.

        if (newSudokuArray != null) //if the a valid value is found for the next cell,
            {return newSudokuArray;} // that valid is returned and stored in the SudokuArray.

        else
        {SudokuArray[row][col] = 0;} //otherwise that cell is set to zero and we keep looking for a valid number (while loop).

        }

    return null;
}
```

The algorithm starts by checking if the cell counter has exceeded 80, meaning the sudoku is ready. If so, it returns the SudokuArray. Otherwise, the generation part of the algorithm begins by creating an arraylist and populating it with 1-9. This list will serve as an indicator and pool from which the next value for each cell is picked. It is important to shuffle the arraylist, which I did using the Collections class, an existing java class. This is done so that a random value is picked from the pool of available numbers each time. The current row and column methods are obtained using a mathematical function I sourced from a website called allegro.cc. The ValidVlue method picks a random value from the arraylist and checks whether it is valid forthe current cell using the CheckBox, CheckRow, and CheckColumn methods. If it isn't, null is returned, and we keep looking for another value. But if it is, that value is stored in the current position of the SudokuArray. The main logic behind the algorithm, or what makes it work, is the use of recursion. Now, the method recursively returns itself with an increase of 1 in the cell counter, and stores the returned value in a "new" 2D array which is essentially actually the same

array. If the returned value doesn't equal null, meaning it is valid for the next cell it is returned again and so the process continues until cellnumber == 81 is reached.

Some more algorithmic thinking and the use of external classes can be seen in the MakeGameBoard class which creates a 2D array with some unsolved cells according to the difficulty level chosen:

```
public static int [][] EasyBoard() //generates a relatively easy board. Desired number of cells left unsolved is approx 20.
{
    int array[][]  = new int[9][9];
    fullSudoku = SudokuGenerator.MakeSudoku(0, array); //fully solved sudoku board is generated

    Random generator = new Random();

    for (int r=0; r<9; r++)
    {
        for (int c=0; c<9; c++)
        {
            int number = generator.nextInt(4); //generates a random number from 0-4
            if (number==1) //the probability of that number equaling 1 is 0.25
            {
                gameArray[r][c]=0; //if this is the case, this index in the gamearray is changed to 0, which will later make it 'unsolved'
            }
            else
            {
                gameArray[r][c]=fullSudoku[r][c]; //otherwise this index in the game array is given the value of the fullsudoku array, mea
            }
        }
    }
    return gameArray; //should generate a board with approx 20 unsolved cells since 20/81 approx equals 0.25.

}
```

The code above shows the EasyBoard method of the MakeGameBoard class which is called if the level easy is chosen. As can be seen a 2D array called fullsudoku is filled using the SudokuGenerator class. Now, a random generator is initiated, which makes use of the external class java.util.Random. Once the nested for loop is entered which iterates though the 9x9 gameArray, a random number ranging from 0-4 (excluding 4) is generated. The probability that this number equals 1 is 0.25 and if this does occur, the current position in the game array will be set to zero, or in other words will be set unsolved. Essentially, since the probability of obtaining 1 from the generator is 0.25 and there are 81 cells in total, approximately 20 cells should end up unsolved, making for a relatively easy sudoku puzzle to solve.

Reading and Writing from Text Files

My program has the option to save a Sudoku in progress for a later time, and then retrieve it. I do this by utilizing the external classes Scanner, File and PrintWriter.

The Save and SavedSudoku methods which write and read to a text file respectively, both also throw a FileNotFoundException, conveniently letting the user know if there might be a problem with the location/name of the file.

This method iterates through the 2D array of textfields and checks whether there is any value inputted by getting the length of the string. If the length equals zero, the printwriter writes “0” to the file, indicating that cell is currently empty. Otherwise, only if the cell isn’t uneditable, which means it was initially solved, it is printed to the file. If the cell is editable but contains a value, which is the last else statement, it means it was filled out by the player. To distinguish this later the value if written to the file with -1 before it. That way, when retrieving the saved sudoku, the SudokuFrame class knows not to make the cells which hold values preceded by -1 have a gray background, as that would indicate they were initially solved when the sudoku was first created.