

**School of Engineering**

**Electrical and Computer Engineering Department**

**Design and Development of a Multi-Player Minesweeper Game**

*A project submitted*

*in partial fulfillment of the requirements for the EEE210 course in SEng*

**by**

Akzhol E. Baktiyar (201593835)

Nurkhan K. Issin(201697711)

**Evaluated by**

Aresh H. Dadlani

**Spring 2019**

**ABSTRACT**

*The Minesweeper project was done in order to get more familiar with how Applications are written. Moreover, the methods and approaches that were learned during the Software Engineering course are implemented in the project. The Level A part of the project was performed successfully. Since this part was not graphical, the flags were not included. Levels B, C and D satisfies basic requirements and rules of the game. JavaFX library used to create those levels. Thus, by working on project assignment OOP coding skills were improved, because of challenges in this project’s journey. However, the main objective of the project was working in a team and getting acquainted with developing application.*

Table of Contents

1. Introduction i
2. Body i

2.1 Level A i

2.2 Level B ii

2.3 Level C iii

2.4 Level D v

Figure3. UML diagram for Level D vi

1. Major Contributions vi
2. Conclusion vii

References viii

# Introduction

Interestingly, Minesweeper was initiated and first written by new-comers to the industry Microsoft in 1990. Motivation to create such game was to encourage consumers to get familiar with the mouse. The main objective of the project is to develop classical Minesweeper game with corresponding principles, which can be understood easily, with further multiplayer extension. Lack of knowledge of JavaFX or Swing is main limitation of this project.

# Body

# 2.1 Level A

The Level A part of the project was needed to display the game in the command window of the programming language.

*2.1.1 Algorithm*

The code initially had two double arrays ***array*** and ***output***, respectively, one of which was created to save the number of bombs surrounding of that element and the bombs themselves, and the other one for displaying to user.

1. The bombs were generated randomly in ***array***.
2. In ***output*** same symbols were assigned to them.
3. The numbers of bombs surrounding the element was counted in ***array***.
4. The inputs were taken from the user.
5. And depending on the position that was chosen the values of the ***array*** was assigned to the ***output***.
6. After each user input, the code checks whether the input position of ***output*** had the bomb or not. If so, the program informs the player that the game over.
7. After each user input, the code checks whether the positions of ***output*** elements having primary symbols are the same as the positions of ***array*** elements including bomb or not. If so, the program informs the player that he/she won. If not goes to the *step 4.*

# 2.2 Level B

For this part of the project the game was needed to be implemented graphically. It was done with the help of the JavaFX.

*2.2.1 Algorithms*

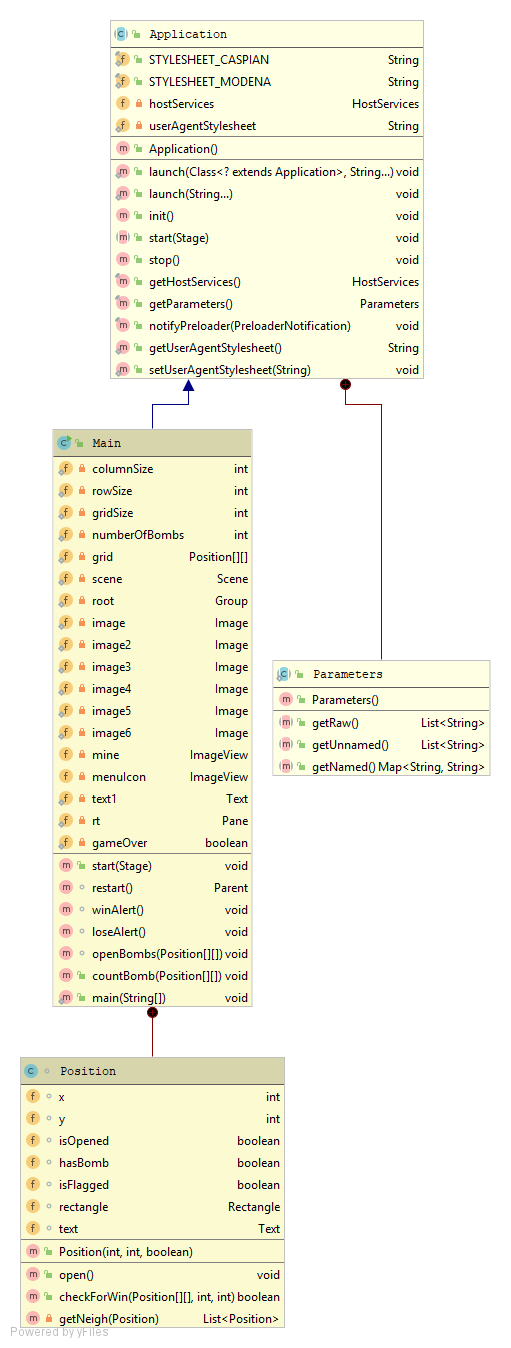
1. The double array of Position class called ***grid*** was created in ***restart()*** method. Moreover, in this method all the features such as number of bombs, retry button and menu were implemented.
2. Pane rt was created and each Position element are added to it, each Position has rectangle, text and perhaps mine. The number of bombs nearby the element was counted with the help of ***countBomb()*** method.
3. If the right button of mouse was clicked rectanglethat was chosen became transparent, for respective ***grid*** element by the method ***open()*** , ***grid.isOpened*** variable was set true.

* If Position has no bombs nearby the ***getNeigh()*** method was called . Thus, it going recursively calling the method ***open()*** for neighbors bound to this element by ArrayList.

4. If the left click of mouse was clicked the image of flag was set on the rectangle chosen. If left button of mouse was clicked one more time the flag was removed. In addition, Position element cannot open if it is flagged.

5. The game was played until left clicked element has bomb or when player wins the game, which is checked by the method ***checkForWin()*** giving affirmative response. If right clicked element has bomb, the ***losealert()*** method was called. More specifically, among all non ***grid.isOpened*** elements this method checked if ***grid.hasBomb*** was set true and the winalert() method was called. Otherwise, all steps beginningfrom the step 3 was repeated again.

6. After the won or lose, the game can be renewed by clicking the **retry** button or by choosing the **New Game** in the menu button.

*2.2.2 UML*

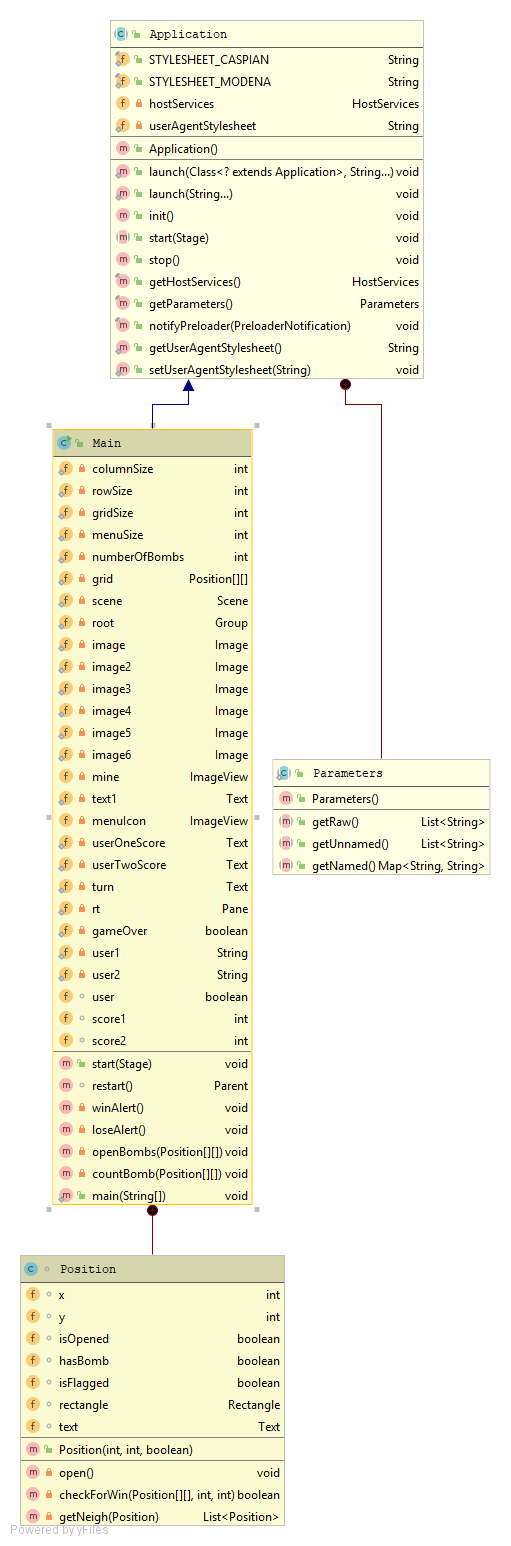
*Figure1. UML diagram for Level B*

# 

# 2.3 Level C

*2.3.1 Algorithms*

As Level C extends Level B, only new content was mentioned in this section. GridPane with textfields were created to enter usernames. Turn is identified by Boolean variable user reversed when primary button of mouse is clicked. Score was added whenever element of class Position opened. Furthermore, in the upper section of scene changeable text was created, which indicated whose turn at the moment. In addition, HBox was added to the lower section of scene which shows scores of players, which updates whenever open method is called.

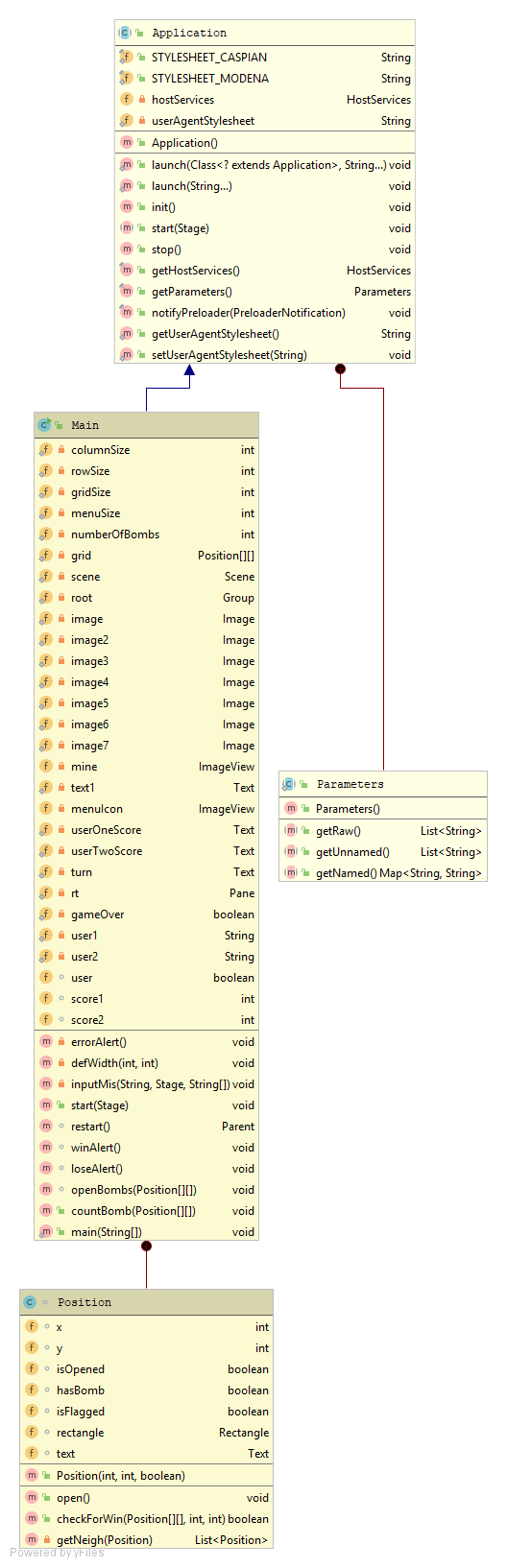
*2.3.2 UML*

*Figure2. UML diagram for Level C*

# 2.4 Level D

*2.4.1 Algorithms*

Only new content was mentioned, because Level D is extension of Level C with some extra features. TextInputDialog was created to enter numbers of rows and columns, if nothing was entered it exits the application, otherwise grid is created with mentioned number of rows and columns. Method defwidth was created to configure size of scene based on number of rows and columns. Input exception is handled using inpuMis method with Alert called inside of method errorAlert.

*2.4.2 UML*

# *Figure3. UML diagram for Level D*

# Major Contributions

**Level A.**

**Akzhol**:

void display(String[][] array) displays array initialized with empty square created to further modify its elements when selected.

void Bomber(String[][] array) randomly places String for bomb.

main method run the game using while loop until win or lose and handles input exceptions.

**Nurkhan**:

void countBomb(String[][] isBomb) method which takes two dimension String array as argument and counts nearby bombs of “unbombed” element and number are assigned to the element.

boolean checkForWin(String[][] isBomb, String[][] arrDisp) checking whether the player won the game or not by comparing “unbombed” elements of arrays.

void search(String[][] isBomb, String[][] arrDisp, int m, int n) handles situation when selected element is empty, without bombs nearby and inserting nearby to the displaying array. If neighboring element is also empty, recursion is used.

**Level B**

**Akzhol:**

Design of Position class elements, stage, scenes, adding menu and additional features within application. Position class which represent one element of grid and implements properties whether bombed or opened and handles right-click and left-click mouse events and implements cases when player loses, wins. In addition, flag behavior is created similar to the original game.

**Nurkhan**:

List<Position> getNeigh(Position) creates arrayList to bind element with its neighbor to further usage to case when empty Position is clicked.

boolean checkForWin(Position[][] element, int row, int col) check whether player wins or not. Condition for win is when all unopened position is consist of only elements with bomb.

**Level C**

Separate scene with keyboard input of usernames with further switching to the main scene, in which actual game can be played, was designed and implemented by Nurkhan. Furthermore, potential exceptions were handled. Calculation and adding of players’ score and who’s turn added to the pane in the scene was done by Akzhol.

**Level D**

Function deciding scene size for different cases was written by Nurkhan. In addition, he added dialog window in which row and column size can be configured in the beginning of the application.

# Conclusion

To conclude, a lot of problems encountered referring to the designing of the application usage JavaFX, due to the lack of some additional knowledge. Furthermore, some issues appeared on implementing classic minesweeper game principles, such as number of flags can be set, clicking in empty position or writing winning and losing cases etc. Specifically, displaying updated computed user scores was accomplished by renewing its content every time position opened. Some problems were occurred handling exceptions when column, row and usernames were inputted. More particularly, for entering column and row separate scene was created, as a result, initial values of written in scene were scene for actual game was created. Mentioned problem was resolved by creating dialog window using Alert class in actual game’s scene.

# References

1. “The most successful game ever: a history of Minesweeper” *Elevator History - Facts and Information*, www.techradar.com/, Accessed on: 21.04.2019