

# **ALGORITHMS & DATA STRUCTURES**

## **COURSE PROJECT REPORT**

The background of the page features a large, light blue watermark of the Hochiminh City International University (HCM-IU) logo. The logo is circular, with the text "HOCHIMINH CITY INTERNATIONAL UNIVERSITY" around the top and "HCM-IU" at the bottom. In the center is a stylized emblem consisting of a circle above a shield-like shape.

### **MINESWEEPER**

**Lecturer: Tran Thanh Tung**

**Date: June 04, 2021**

**Semester 2, Academic year 2020 - 2021**

# Project DSA – Topic: Minesweeper

## \*Team members and contribution

Full name	Student ID	Contribution (%)
Nguyễn Hoàng Linh	ITITIU19023	25
Trần Hoàng Long	ITITSB19004	25
Nguyễn Đức Minh	ITITIU19030	25
Trần Quang Tùng	ITITIU19237	25

## \*Tasks

Task	Name
Create user interface	Minh
Handle events	Minh, Tùng, Long
Handle flow of the app	Linh
Manage API	Linh
Advanced features	Long
Leaderboard	Tùng

## \*Github link

<https://github.com/Lucky-Star-999/Minesweeper>

# Table of Contents

<b>I. Introduction</b>	5
1. What is minesweeper?	5
2. Objectives	5
3. Methods	5
<b>II. Game rules</b>	6
<b>III. API</b>	8
<b>IV. Class diagram</b>	9
1. API	9
2. Main app	10
<b>V. Minesweeper UI</b>	11
1. Objectives:	11
a) User Experience:	11
b) User Interface:	14
2. Design using Adobe XD	16
a) Why we chose Adobe XD:	16
b) Process:	16
<b>VI. Data structures and algorithms</b>	18
1. Data structures	18
1.1. Two-dimensional array	18
1.2. One-dimensional array	19
1.3. Stack	19
1.4. Queue	19
2. Algorithms	20
2.1. Time counter	20
2.2. Self-adjust number of bombs	21
2.3. Checking if player loses	22

2.4. Checking if player wins.....	22
2.5. Undo feature .....	23
2.6. Creating random bombs .....	23
2.7. Linear Search .....	24
2.8. Insertion Sort.....	25
2.9. Middle click function.....	26
2.10. Expand all empty function.....	27
<b>VII. Design patterns.....</b>	<b>28</b>
1. Singleton Pattern.....	28
2. Factory Pattern .....	29
<b>VIII. Conclusion.....</b>	<b>30</b>
1. Experience .....	30
2. Problems .....	31
3. Recommendations .....	31
4. Future improvements.....	31
<b>IX. References .....</b>	<b>31</b>

# **I. Introduction**

## **1. What is minesweeper?**

Minesweeper is a logic puzzle game that takes place on a grid "minefield". The player's goal is to clear (open) every square in the grid without hitting the mine - and to do it as quickly as possible. When you open the squares, clues will also appear as numbers representing the number of mines located in the 8 squares around them.

During the game, you can mark places where you think there will be mines by right-clicking to insert a flag. Be careful, just accidentally hitting a landmine is enough to make the game over!

## **2. Objectives**

The goal for our team is to create the game minesweeper with knowledge applied from Data Structure and Algorithm course as Search, Sort, Stack, Queue, Breadth First Search, etc.

In addition, we want to apply knowledge of website to this project to make the app can be published on the website so that everyone can play and especially create the API to support the app to be runnable in any platforms without hard effort.

## **3. Methods**

Our Minesweeper game is written in JavaScript, HTML and CSS. We also use Node.js for the backend. We also put the project onto a website so that anyone can enjoy. This is the website for our project.

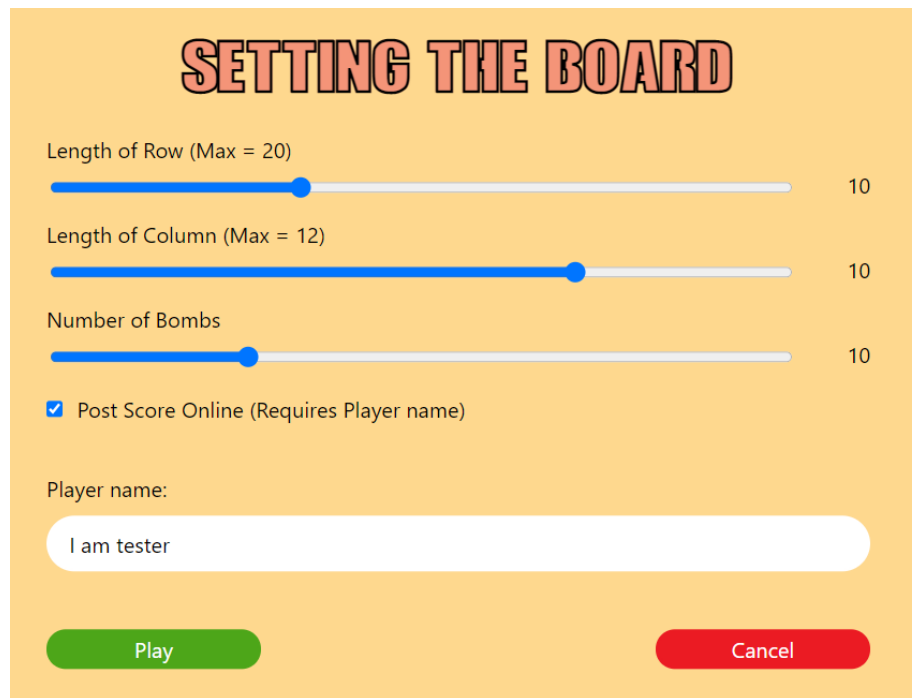
<https://linh-iti19023-minesbackup.herokuapp.com/>

In addition, to create the challenge for the project, we also created an API to support the project. The function of this API is to create a 2D array containing information of all cells in a square grid of the game, for example cells containing bomb will have the value -1, if there are 7 bombs around a cell, the cell will have the value 7. Here is our API link

<https://myapi-minesweeper.herokuapp.com/>

## II. Game rules

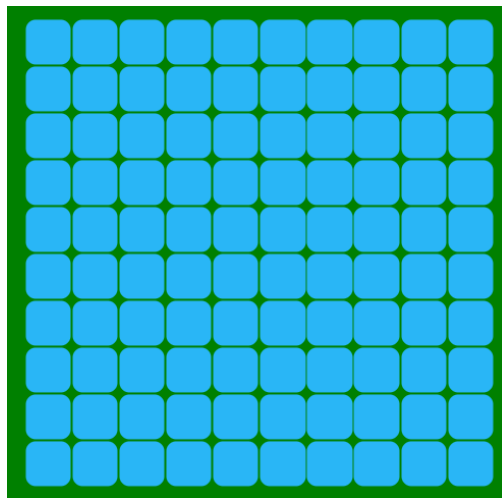
First, the player needs to choose the dimensions of the grid and number of bombs. Then if the player wants to save the score, just click on the box “Post Score Online”



The screenshot shows a settings interface titled "SETTING THE BOARD" in large, stylized, pink-outlined letters. Below the title are three sliders: "Length of Row (Max = 20)" with a value of 10, "Length of Column (Max = 12)" with a value of 10, and "Number of Bombs" with a value of 10. Each slider has a blue handle and a numerical value displayed on the right. Below the sliders is a checkbox labeled "Post Score Online (Requires Player name)" which is checked. Underneath is a text input field labeled "Player name:" containing the text "I am tester". At the bottom are two buttons: a green "Play" button and a red "Cancel" button.

*Figure 1: Board setting*

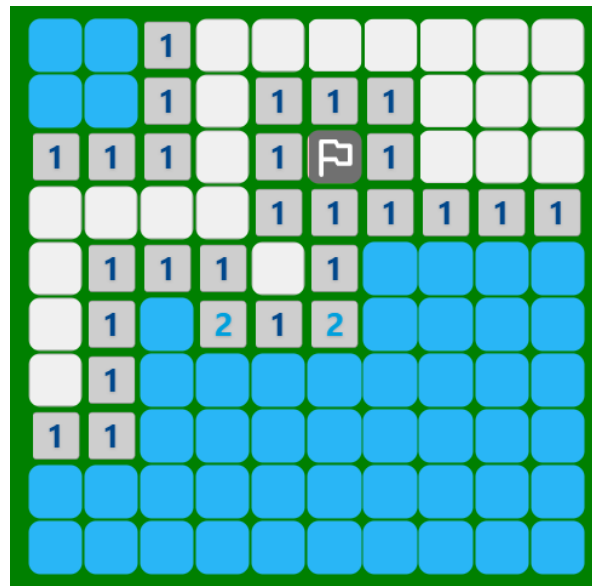
Then the grid will show on the screen



*Figure 2: The grid*

The grid has many cells, and there are many bombs placed behind some cells. The player has to carefully open as more as possible cells by clicking on them. However, if the player clicks on a cell with a bomb behind, he will lose.

Each cell will have a number, and it tries to tell the player that around this cell (3x3 area) has how many bombs. For example, we have surrounding the flag has 8 cells, and each of them are all number 1, it means around each 8 squares has 1 bomb. And we easily conclude that the cell which contains flag has a bomb inside.



*Figure 3: Example for cells*

The player can put on a flag on each cell by using right click to mark the cell which has the most possible for a bomb inside.

If the grid remain only cells are not opened that have bombs inside, the player will win.



*Figure 4: Player win*

The special of this game is that when clicking on the grid first, the player cannot lose. This is because we create the appearance of the grid first, but have not create the bombs inside yet until the user clicks on the grid at the first time. So, players can feel free to play this game without worrying about clicking on bombs at the first time.

### III. API

When a website requests data from our API, that API will return a 2D array containing the necessary information.

```
[[0,0,0,0,0,1,2,-1,2,1],[1,1,0,0,0,1,-1,3,-1,1],[-1,2,0,0,0,1,1,3,2,2],[-1,2,0,0,0,0,0,1,-1,1],
[1,1,0,1,1,1,0,1,1,1],[0,0,0,1,-1,2,2,1,1,0],[0,0,0,1,2,-1,2,-1,1,0],[1,1,1,0,1,1,2,1,1,0],
[1,-1,1,0,0,0,0,0,0,0],[1,1,1,0,0,0,0,0,0,0]]
```

*Figure 5: Example of data returned from API*

To be able to control this API, users need to operate directly on the following URL:

<https://myapi-minesweeper.herokuapp.com/>

Our full URL consists of 4 components



<https://myapi-minesweeper.herokuapp.com/<a>/<b>/<c>/<d>>

which,

a: the first place the player clicks on the square grid of the game; the goal is to avoid touching bomb at the first click of player

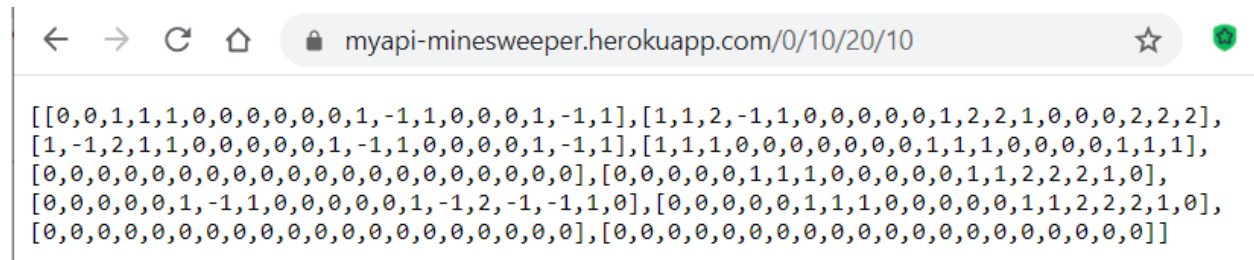
b: number of bombs in the game

c: number of cells in a row (length of grid)

d: number of cells in a column (height of grid)

For example:

<https://myapi-minesweeper.herokuapp.com/0/10/20/10>



*Figure 6: API response*

The first position that the user clicks on is position 0, the number of bombs is 10, the length of the grid is 20 cells, the height of the grid is 10 cells. The first position that the user clicks is converted to 1D. For example, our grid is 10x10 (length is 10 cells, height is 10 cells), so points at 1D coordinates will be marked from 0 to 99

## IV. Class diagram

### 1. API

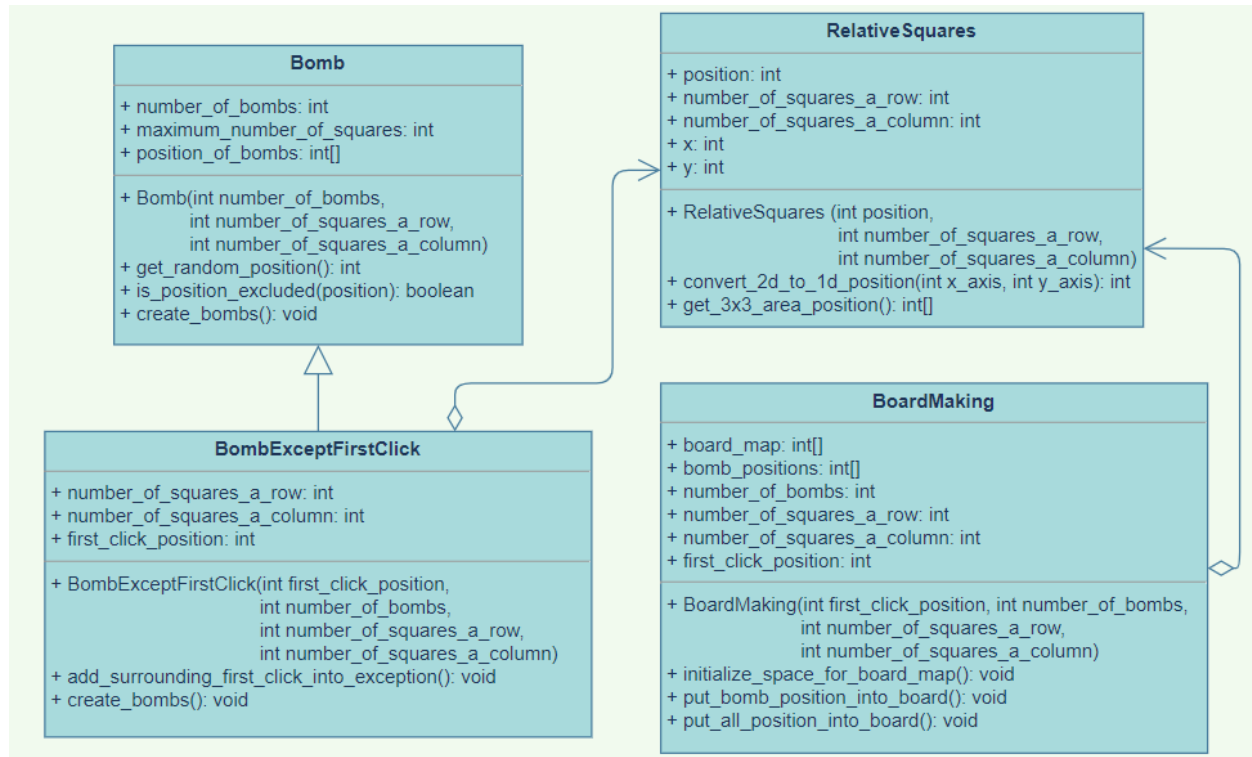


Figure 7: Class diagram for API

## 2. Main app

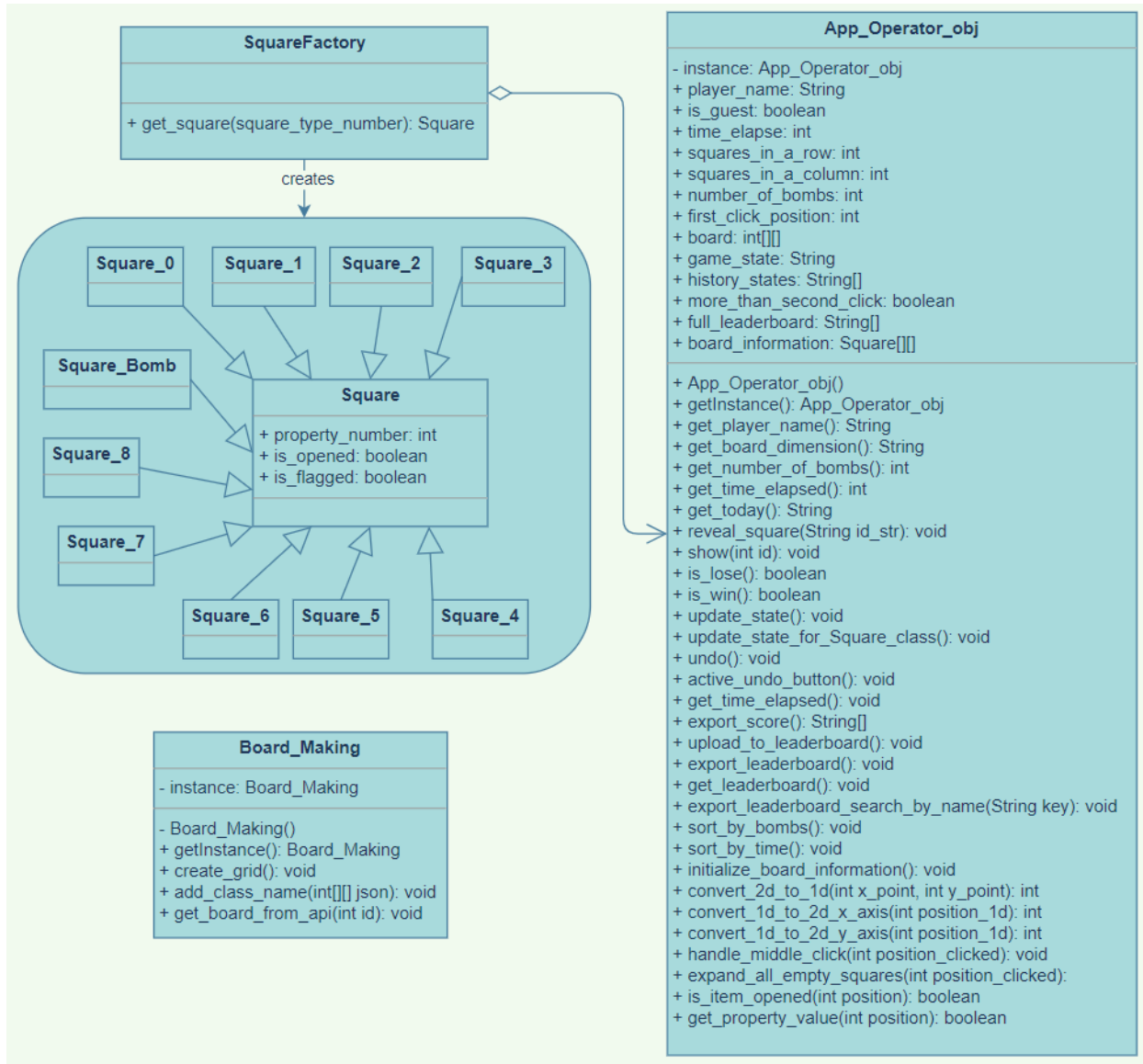


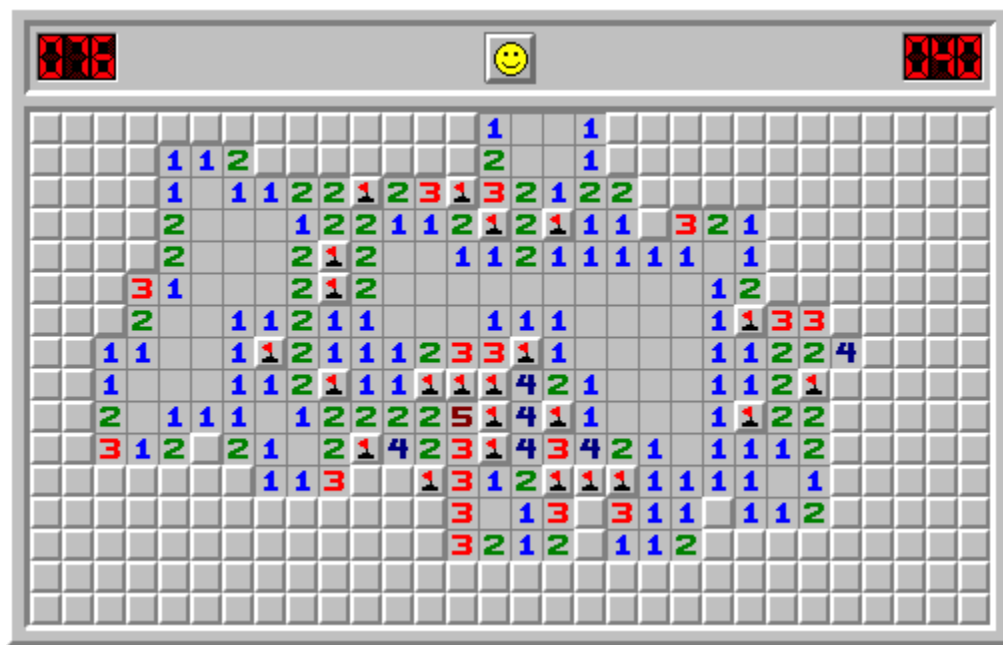
Figure 8: Class diagram for main app

## V. Minesweeper UI

### 1. Objectives:

#### a) User Experience:

Minesweeper started out as a simple game available for all models with Windows installed. With the sole objective of discovering all “safe” spots, the overall design of the game is naturally minimal.



*Figure 9: Window's Minesweeper*

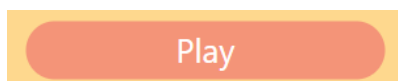
As seen from the figure above, there are simple UI for points, clock, an icon to indicate your game's status, and a playing field. It was adequate, for its main objective.

However, time brings change and users (or players) now require more beautiful and sophisticated experience during interaction with computers. So, what started out as a simple and adequate product in the late 80s have now turned into challenges for present developers: simplicity.

With such a linear objective, how is it that we can make it better, more enjoyable, help others feel better to look at the screen and think?

As a matter in fact, there are actual methods to improve the user experience. Some of which contains Interaction Reward, Color Management, Detail Emphasis, and Customizability.

When using computer, it is important to reward users upon their interactivity. The reason being that, it helps users know what they are doing, and it is pleasing to the eyes to see the machine reacts to actions. Therefore, we implement animated buttons and play grids.



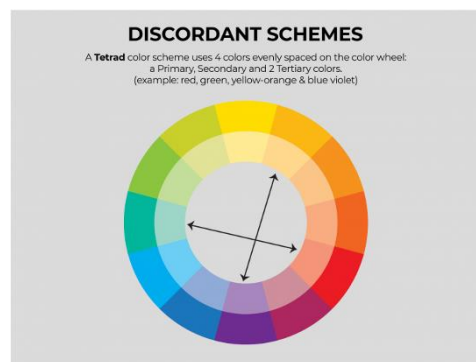
*Figure 10: “Play” button – normal*

*Figure 11: “Play” button – hover*

Minesweeper is also a thought-provoking game, so it is feasible to make the interface easier to stare at for an extended amount of time. In addition, there is also a need for color consistency, as a general rule of design. For this, a color wheel is implemented. In order to not affect the users’ eyes strongly, pastel-like colors are used most often, while important features have primary colors for highlights.



*Figure 12: Setting screen’s button have stronger colors as opposed to light-yellow background*



*Figure 13: Color Wheel*

Detail Emphasis is also an important aspect in any design. These highlights act as guidelines to control the attention of users. Equally important, is the de-emphasis of details. This would, in turn, turns the user’s attention away. For example, we want to encourage players to continue on playing during the game and not quit. Consider these two scenarios:



*Figure 14: De-emphasized “Exit” button*



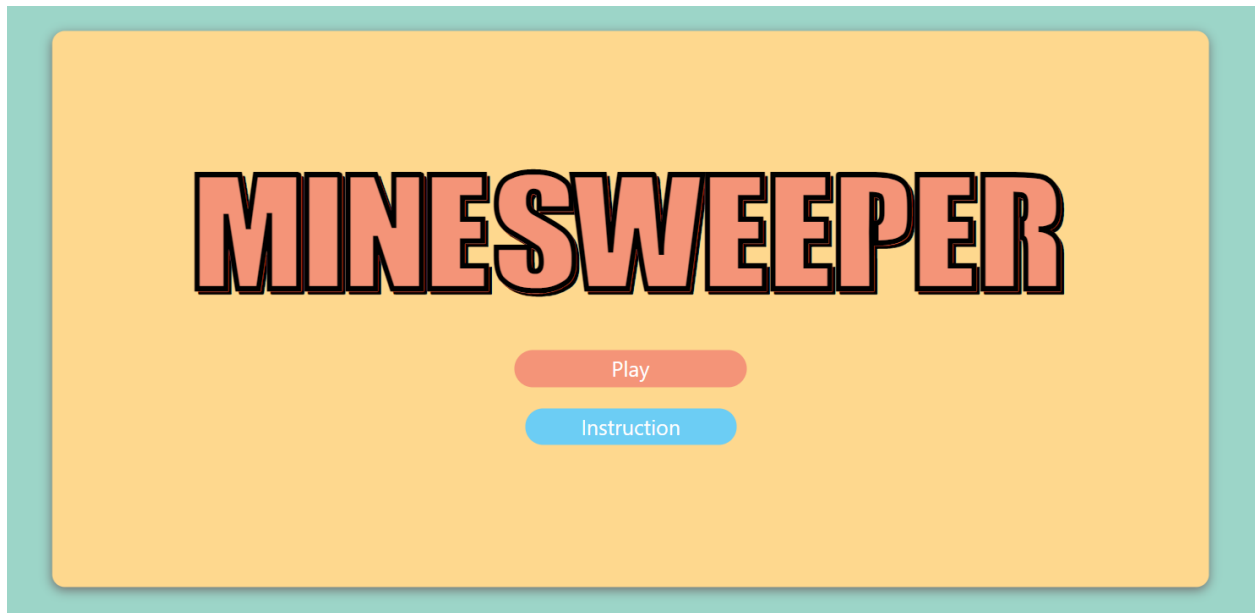
*Figure 15: “Exit” button made similar to the other buttons*

It is clear that Figure 5 lightens the possibility that players can quit at any time. That, in combination with the color red is of the more noticeable colors.

Last, but not least, is Customizability. The original Minesweeper had more limited choices for players when it comes to game difficulty. In our project, we let players have full control of their game, increasing and decreasing difficulty as needed. There is also a leaderboard, for those wanting to brag about their level. Figures for Customization can be seen in the next section, Settings Screen.

#### **b) User Interface:**

Resulting from the aforementioned concerns when dealing with User Experience, the product UI is somewhat minimal, but hopefully it reaches to the standards of today’s front-end requirements.



*Figure 16: Home Screen*

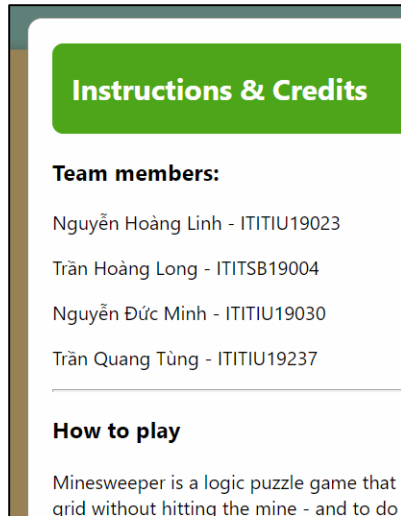


Figure 17: Instructions and Credits available

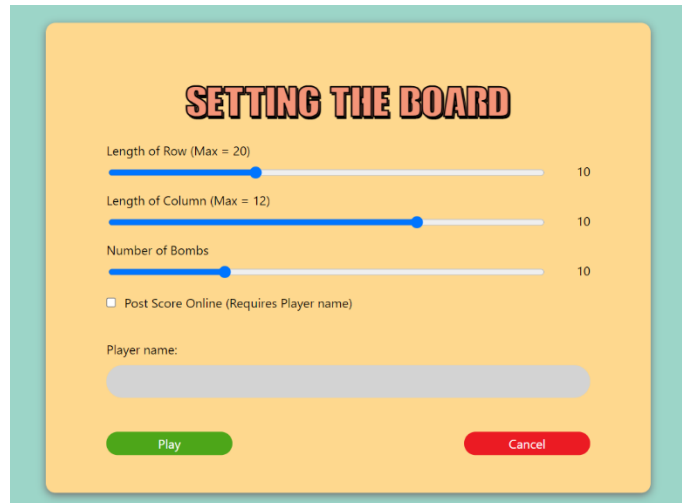


Figure 18: Settings screen

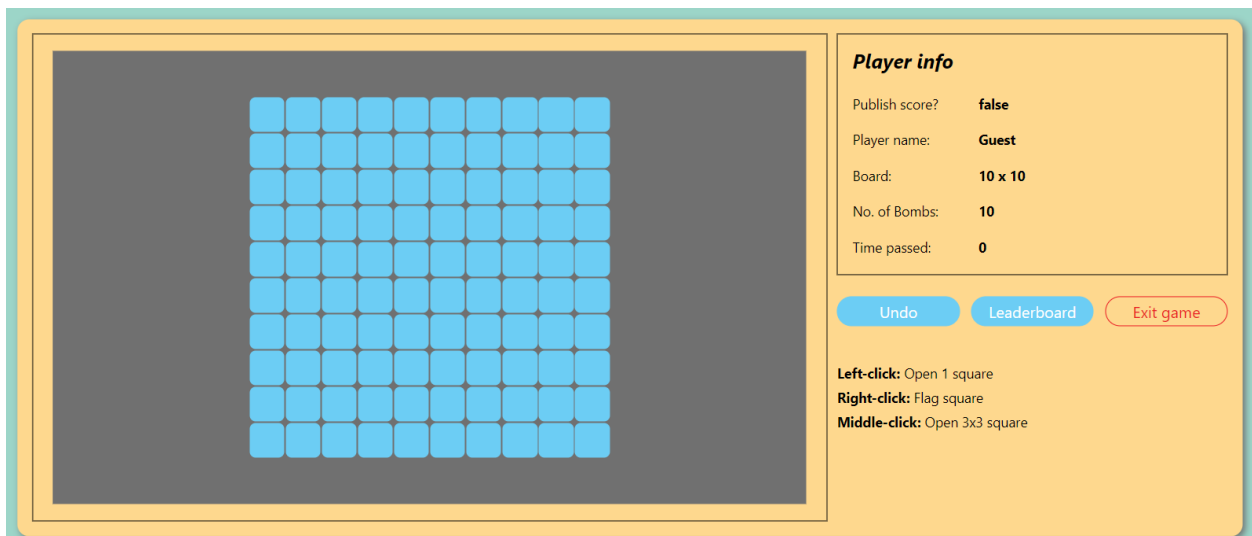
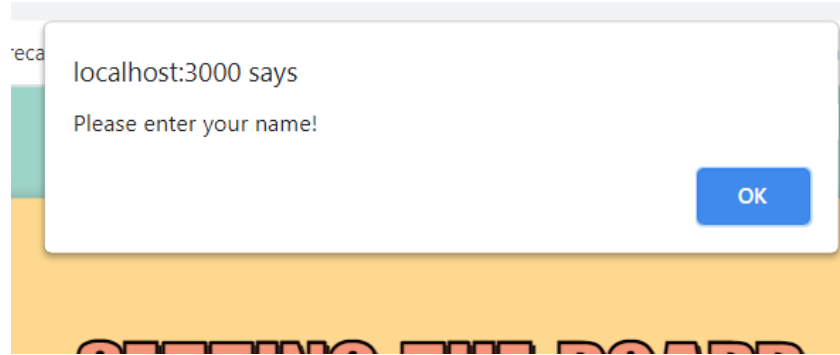


Figure 19: Game Screen

Player information on the right is displayed dynamically depending on what was configured in the Settings Screen.

There are also validations to require players to follow in order to play. One instance of which is the alert of a player checking the “Post Score Online” box, but refused to give a player name:



*Figure 20: Google Chrome alerts player to enter his name*

## **2. Design using Adobe XD**

### **a) Why we chose Adobe XD:**

Adobe XD is a design tool from Adobe, mostly used for website design. Its simplicity, easy to understand and memorize functionalities can export all kinds of websites from the simplest to the most complicated ideas.

For the project, we planned to make a webapp, so it is fitting to use Adobe XD to sketch out the website prior to coding the front-end of the app. Here, we tested colors, fonts, sizes, and positions of components. It can be seen as drawing up a blueprint before actually building the product. Furthermore, Adobe XD provides possibilities of wireframes, so that we can see how each component would react when hovered over, pressed, changed...

### **b) Process:**

We started out by looking over designs from other developers. Of course, the original of which is Window's Minesweeper.

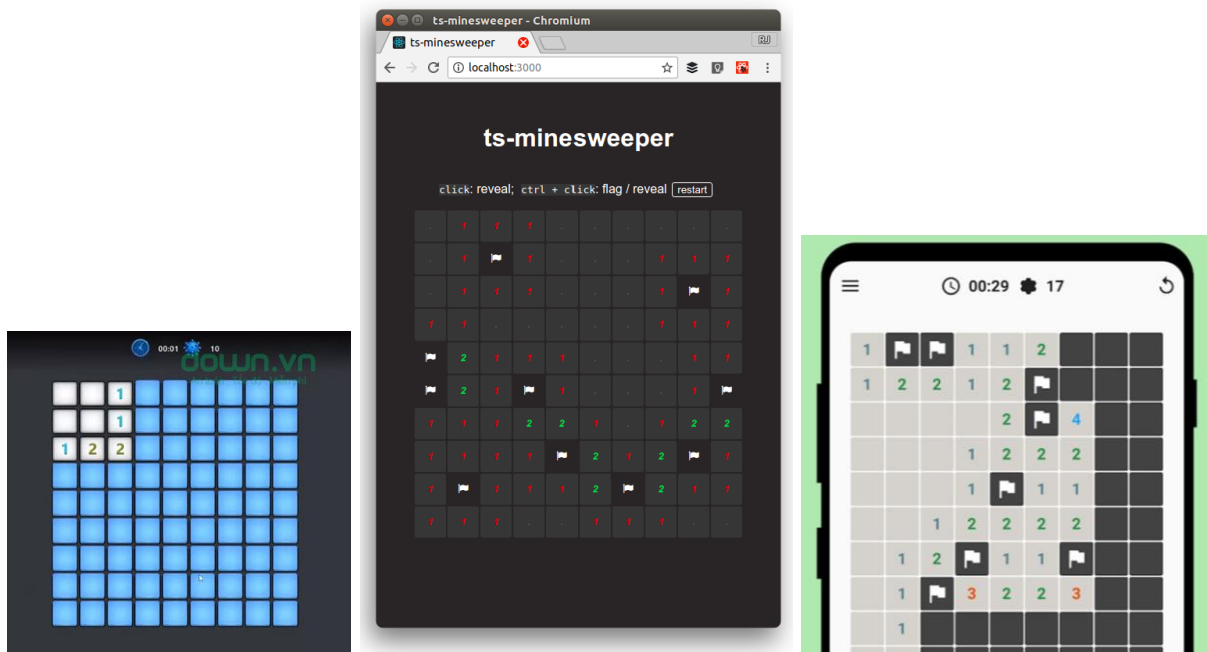




Here, we can see that the playing field is quite simple. It is not monotonous, but the scaling of the grid is rather small, which may tire the player's eyes. It was a challenge to make something repetitive interesting and not boring.

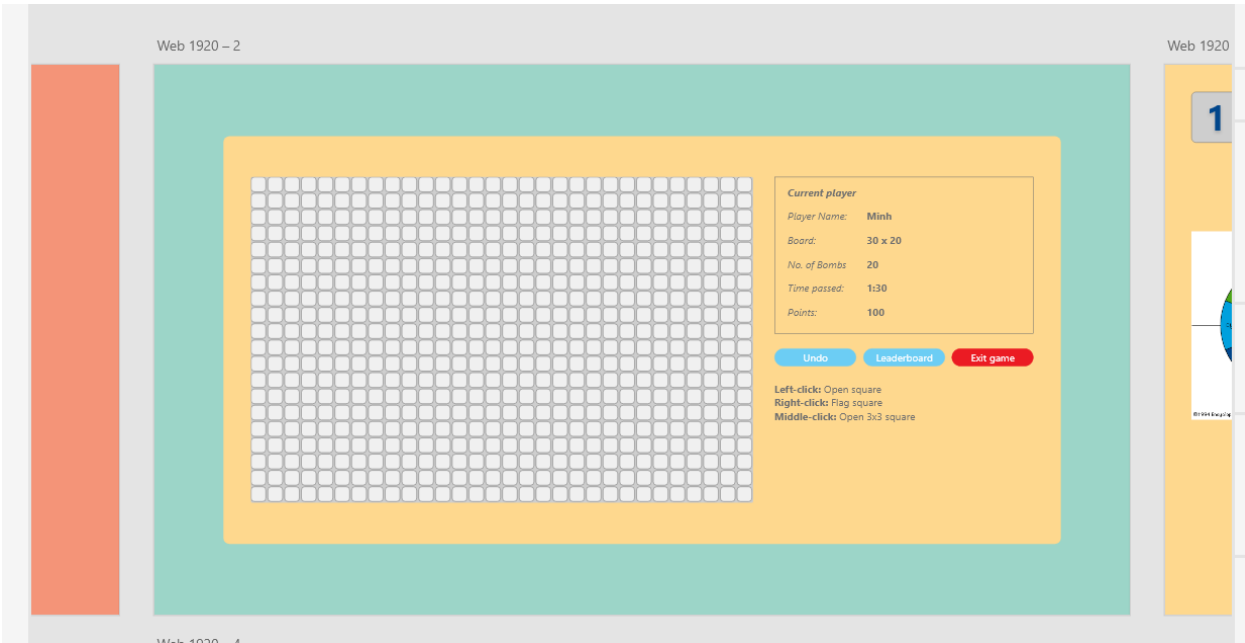
*Figure 21: Window's Minesweeper*

Over time, designs for the game have taken place, and we found that many of which are simple and comprehensive. We learned from them.



*Figure 22: Examples of Minesweeper designs*

From which, we chose our pastel colors as the base of our interface, and draw everything out on XD. This gives us a clear picture of what the product looks like, accurate to the pixels.



*Figure 23: Adobe XD design*

## VI. Data structures and algorithms

### 1. Data structures

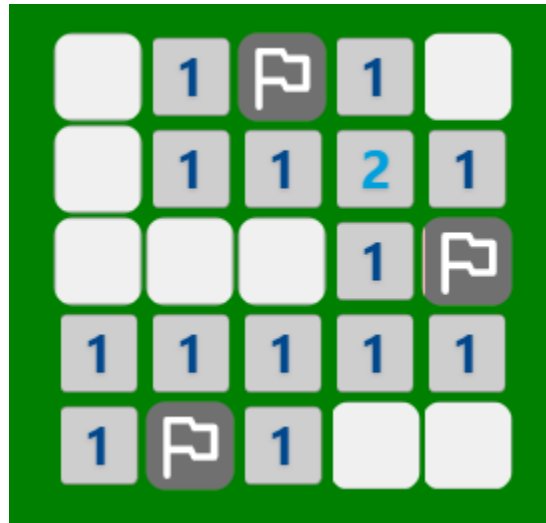
#### 1.1. Two-dimensional array

We use two-dimensional (2D) array to store some important data as information of the grid. For example:

We have the 2D array as:

```
[
  [0, 1, -1, 1, 0],
  [0, 1, 1, 2, 1],
  [0, 0, 0, 1, -1],
  [1, 1, 1, 1, 1],
  [1, -1, 1, 0, 0]
]
```

With the array, we can generate the grid with all the numbers behind each cell, as the following figure



*Figure 24: Graphical of the 5x5 grid*

## 1.2. One-dimensional array

To make it easier to deal with cells in the grid, we convert the coordinates of the cells from 2D to 1D. Then we put all these 1D coordinates into a one-dimensional array. Or sometimes, we need a function to display cells in a 3x3 range when the player use middle-click, this function will return a 1D array containing the information of the cells that need to be opened.

## 1.3. Stack

To be able to create the Undo move feature, we need to use Stack. With JavaScript, we can use 1-dimensional arrays as an alternative to Stack. Whenever we need to “peek” value, we will manipulate the last index of the array. When we need the “pop” value to revert the previous stage of the game, we will use the pop() function, which is supported by JavaScript.

## 1.4. Queue

To automatically expand empty cells (not have any bombs around), we use Breadth First Search (BFS). By using BFS, we need a queue to make it easier to

handle. When we need to record some nodes, we will enqueue all the nodes into the queue. When the nodes have been travelled, we dequeue them from the queue.

## 2. Algorithms

### 2.1. Time counter

JavaScript support an interesting function to support for making a time counter, `setInterval()`. The `setInterval()` method calls a function or evaluates an expression at specified intervals (in milliseconds). The `setInterval()` method will continue calling the function until `clearInterval()` is called, or the window is closed.

To make the time counter, we set the interval for 1 seconds, which means each second past, we will update and print out the time on the screen.

However, we need to modify some things before printing out the time on the screen, because we want to display the time with format

`<minutes> : <seconds>`

First, we have a counter which is initialize at value 0, and it will increase one by one infinitely until the user win, lose or the program is terminated. We consider this value as number of seconds elapsed

Next, we convert the seconds elapsed into two parts: minute and second (range is 60)

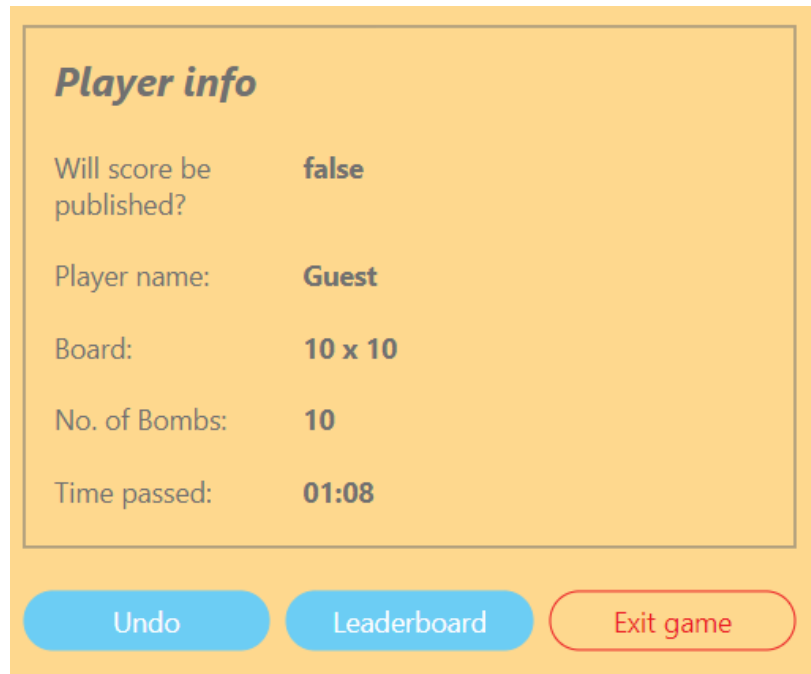
To convert to minute, we divide the seconds elapsed by 60, and we have the minute

`minute = seconds_elapsed / 60`

To convert to second, we calculate the modulo of the result taking from division between the seconds elapsed and 60.

`second = seconds_elapsed % 60`

Then, we print out “minute” and “second” on the screen

A yellow rectangular dialog box with a thin black border. At the top left, the text "Player info" is written in a bold, italicized, dark blue font. Below this, there are five rows of text, each with a label on the left and a value on the right. The labels are "Will score be published?", "Player name:", "Board:", "No. of Bombs:", and "Time passed:". The values are "false", "Guest", "10 x 10", "10", and "01:08" respectively. At the bottom of the dialog box, there are three rounded rectangular buttons. The first two are blue with white text: "Undo" and "Leaderboard". The third is red with white text: "Exit game".

Will score be published?	false
Player name:	Guest
Board:	10 x 10
No. of Bombs:	10
Time passed:	01:08

Undo    Leaderboard    Exit game

*Figure 25: Player info with time passed*

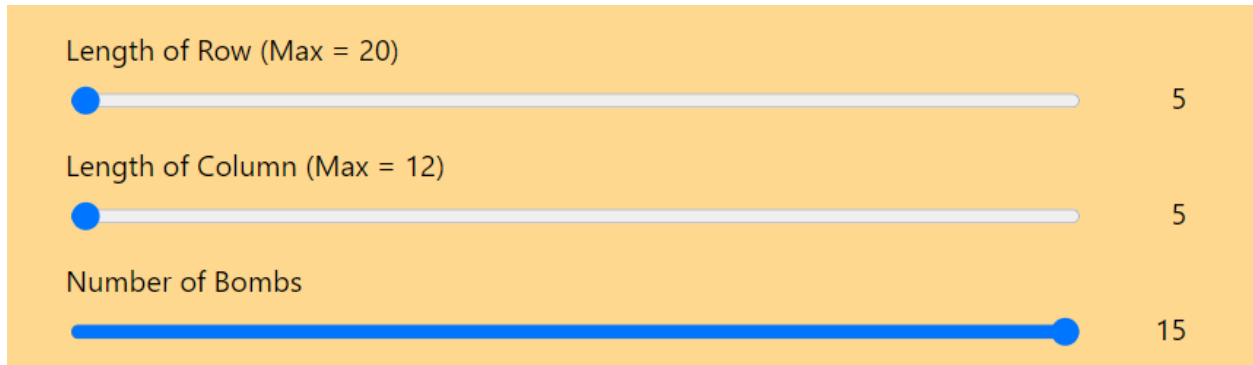
## 2.2. Self-adjust number of bombs

When user choose dimensions and numbers of bomb for the grid, we must make sure that the bombs number is suitable for the board, for example, if the dimension of the grid is 5x5, but user also set the number of bombs is 100, so it will cause many problems.

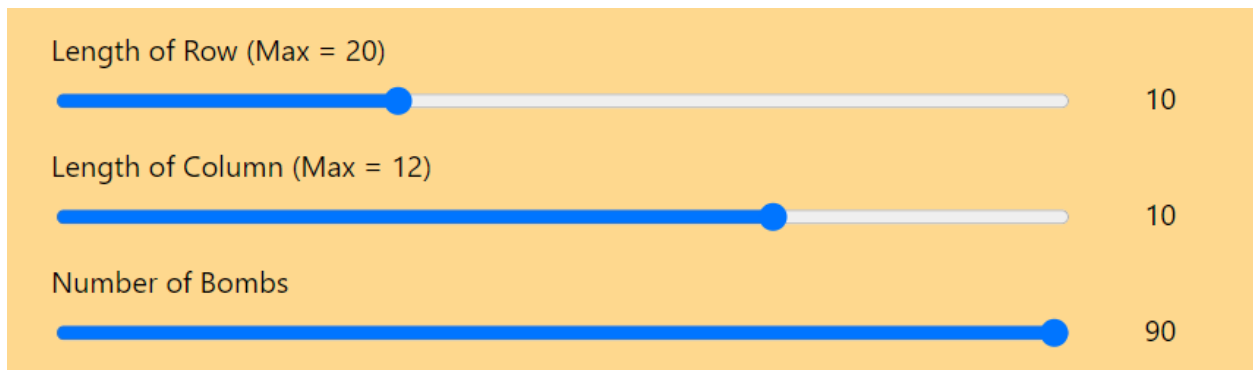
So, we have a formula to solve this problem

$$\text{Max number of Bombs} = (\text{Row length} * \text{Column length}) - 10$$

When there are any changes of length of row or length of column by user, the option for setting number of bombs will be updated. For example, when length of row is 5, and length of column is 5, the maximum number of bombs will be 15. When length of row is 10, and length of column is 10, the maximum number of bombs will be  $10*10-10 = 90$ .



*Figure 26: Board 5x5 with 15 bombs*



*Figure 27: Board 10x10 with 90 bombs*

### 2.3. Checking if player loses

Whenever a player clicks on a cell, the system will update and record all information about the properties of the cell. If the system detects a cell containing a bomb is opened by user, it will throw the announcement that the player loses.

### 2.4. Checking if player wins

We have the formula

$$\text{Cells\_are\_not\_bomb} = (\text{Row\_length} * \text{Column\_length}) - \text{Bomb\_number}$$

which is,

**Cells\_are\_not\_bomb:** Number of cells that do not have a bomb inside

**Row\_length:** Number of cells in a row

**Column\_length:** Number of cells in a column

**Bomb\_number:** Number of bombs

We keep continuously check the information of the grid whenever the user clicked on a cell as how many cells are opened, how many cells are set flag, how many cells are not opened. If the number of cells are opened is greater or equal than the number of cells that are not bomb, the player will win.

## 2.5. Undo feature

Whenever a player clicks on a cell, the system will update and record all information about the cells and put them in a stack. With JavaScript, we can use arrays as a stack.

Array in JavaScript is very special, each element can contain any types of data, even an array. Moreover, array in JavaScript can remove the last elements, so it is the perfect thing to use as a stack.

Assume that we have an array containing all information about the cells as which cells are clicked on, which cells are set a flag, which cells are not clicked on. Every time a player clicks on a cell, all the information will be updated, then be packed into an array, then we push this array into another array, which is called stack.

Every time player wants to undo, we pop out the stack and get the old information. Then we use it to revert the state of all the cells.

## 2.6. Creating random bombs

First, we need to put position of the bombs in the board, then we calculate how many bombs around each cell. However, the board will not be created until a player click on the grid at the first time, because we want the probability for clicking on a bomb by player is 0 percent at the first click.

When a player clicks on a cell, we will get the position, then we handle it such that the surrounding of this cell has not any bombs.

To create the random bombs, we use the random method in JavaScript. By using random, we will create the random positions for the bombs. Before doing that, we need to define maximum and minimum possible positions for the bombs.

For example, the board has dimensions 20x10, which means the length of the board is 20 cells, and the height is 10 cells. So, the possible position is 0 to 199 and we get the maximum as 199, and minimum as 0.

We have the formula for calculating the minimum and maximum possible position

**Minimum position = 0**

**Maximum position = Row length \* Column length – 1**

## 2.7. Linear Search

First, we use `entry()` function to clear all contents in leaderboard and then append element to add several headers into table leaderboard.

Next, we use Linear algorithm. In this type of search, a sequential search is made over all items one by one. Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection.

This function's structure is just 1 simple loop with condition statements to add every piece of data (play\_name, board\_dimension, num\_of\_bombs, date, time\_elapse, game\_state) into `leaderboard_table`. Then append `row_element` into `leadboard_table`.

```
export_leaderboard_search_by_name: function (key) {  
  $("#leaderboard_table").empty();  
  let elements = '<tr><th>Player name</th><th>Board dimension</th>' +  
    '<th>Number of bombs</th><th>Date played</th><th>Time Elapsed (Seconds)</th><th>Result</th></tr>';  
  
  $("#leaderboard_table").append(elements);  
  
  for (let i = 0; i < this.full_leaderboard.length; i++) {  
    if (this.full_leaderboard[i].player_name.includes(key)) {  
      let row_element = '<tr>';  
      row_element += '<td>' + this.full_leaderboard[i].player_name + '</td>';  
      row_element += '<td>' + this.full_leaderboard[i].board_dimension + '</td>';  
      row_element += '<td>' + this.full_leaderboard[i].number_of_bombs + '</td>';  
      row_element += '<td>' + this.full_leaderboard[i].date + '</td>';  
      row_element += '<td>' + this.full_leaderboard[i].time_elapse + '</td>';  
      row_element += '<td>' + this.full_leaderboard[i].game_state + '</td>';  
      row_element += '</tr>';  
      $("#leaderboard_table").append(row_element);  
    }  
  }  
}
```



*Figure 28: Linear Search*

## 2.8. Insertion Sort

In section `sort_by_bomb` and `sort_by_time`, we use insertion sort to serialize time and bombs according to the number of bombs and time after each game

This function's structure includes 2 'for' loops and 1 'while' loop: in command line for with condition statements to validate index values do not go off bounds of subarray, it will run respectively from the first to the last element of subarray; with the second loop while, it will relocate elements having bigger value than "current" to after its original position

Using Insertion Sort with bombs and time will make leaderboard become clear and easy to find the results and rankings after each game.

```
246 sort_by_bombs: function () {
247   let temp_leaderboard = this.full_leaderboard;
248   // Insertion sort
249   let n = temp_leaderboard.length;
250   for (let i = 1; i < n; i++) {
251     // Choosing the first element in our unsorted subarray
252     let current = temp_leaderboard[i];
253     // The last element of our sorted subarray
254     let j = i - 1;
255     while ((j > -1) && (current.number_of_bombs < temp_leaderboard[j].number_of_bombs)) {
256       temp_leaderboard[j + 1] = temp_leaderboard[j];
257       j--;
258     }
259     temp_leaderboard[j + 1] = current;
260   }
261 }
```

*Figure 29: Sort\_by\_bombs*

```
281 sort_by_time: function () {
282   let temp_leaderboard = this.full_leaderboard;
283   // Insertion sort
284   let n = temp_leaderboard.length;
285   for (let i = 1; i < n; i++) {
286     // Choosing the first element in our unsorted subarray
287     let current = temp_leaderboard[i];
288     // The last element of our sorted subarray
289     let j = i - 1;
290     while ((j > -1) && (current.time_elapse < temp_leaderboard[j].time_elapse)) {
291       temp_leaderboard[j + 1] = temp_leaderboard[j];
292       j--;
293     }
294     temp_leaderboard[j + 1] = current;
295   }
296 }
```

*Figure 30: Sort\_by\_time*

## 2.9. Middle click function

This function will reveal all the 3x3 area around the chosen click position with a condition. When the number of flags in surrounding area (3x3 area) is equal to the value of the clicking cell, the middle click action will activate the middle click function. The middle click function will also be activated even there are bombs unrevealed as long as the condition mentioned above is satisfied.

This function's structure is just 2 simple loops with condition statements to validate that the new generated position values do not go off bounds of the game board.

```
let flags = 0;
// counting flags of the surrounding area
for (let i = -1; i < 2; i++)
  for (let j = -1; j < 2; j++)
    if (i !== 0 || j !== 0)
      {
        let new_x = x_clicked + i;
        let new_y = y_clicked + j;
        if (new_x < 0 || new_x >= row_length || new_y < 0 || new_y >= column_length)
          continue;

        let new_pos = this.board_information[new_y][new_x];
        if (new_pos.is_flagged)
          flags++;
      }
```

*Figure 31: Counting number of flags of 3x3 area*

```

if (info.property_number == flags)
{
    for (let i = -1; i < 2; i++)
        for (let j = -1 ; j < 2; j++)
            if (!(i == 0 && j == 0))
            {
                let new_x = x_clicked + i;
                let new_y = y_clicked + j;
                if (new_x < 0 || new_x >= row_length || new_y < 0 || new_y >= column_length)
                    continue;

                let new_pos = this.board_information[new_y][new_x];
                if (new_pos.is_flagged == false)
                    this.show(this.convert_2d_to_1d(new_x,new_y));
            }
}

```

*Figure 32: Open surrounding 3x3 area*

## 2.10. Expand all empty function

This function will be activated if the chosen position has the value of 0 (an empty white cell on the GUI). This function will continue to expand all the adjacent 0 cells until reaches the cells that have value larger than 0.

The algorithm we use for this function is BFS (Breadth-First Search) running on a 2D array. We use a Queue to store the positions that have potential to keep extending. The algorithm will expand level by level to find all potential 0-value cells and put them in the Queue to keep searching others.

For checking which positions are already visited, use a “visited” array to store all the positions that we visited so that we will not revisit those positions again in the method call.

```

expand_all_empty_squares: function (position_clicked) {
    let queue = new Queue();
    let row_length = this.squares_in_a_row;
    let column_length = this.squares_in_a_column;
    let visited = [];

    if (this.get_property_value(position_clicked) == 0 && !this.is_item_opened(position_clicked))
        queue.enqueue(position_clicked);

    while (!queue.isEmpty()){
        let item = queue.dequeue();
        this.show(item);
        visited.push(item);
        if (this.get_property_value(item) == 0)
        {
            let x = this.convert_1d_to_2d_x_axis(item);
            let y = this.convert_1d_to_2d_y_axis(item);
            for (let i = -1; i < 2; i++)
                for (let j = -1; j < 2; j++)
                    if (i != 0 || j != 0)
                    {
                        let new_x = x + i;
                        let new_y = y + j;
                        if (new_x < 0 || new_x >= row_length || new_y < 0 || new_y >= column_length)
                            continue;

                        let new_item = this.convert_2d_to_1d(new_x, new_y);
                        if (!this.is_item_opened(new_item) && visited.indexOf(new_item) == -1)
                            queue.enqueue(new_item);
                    }
                }
            }
    }
}

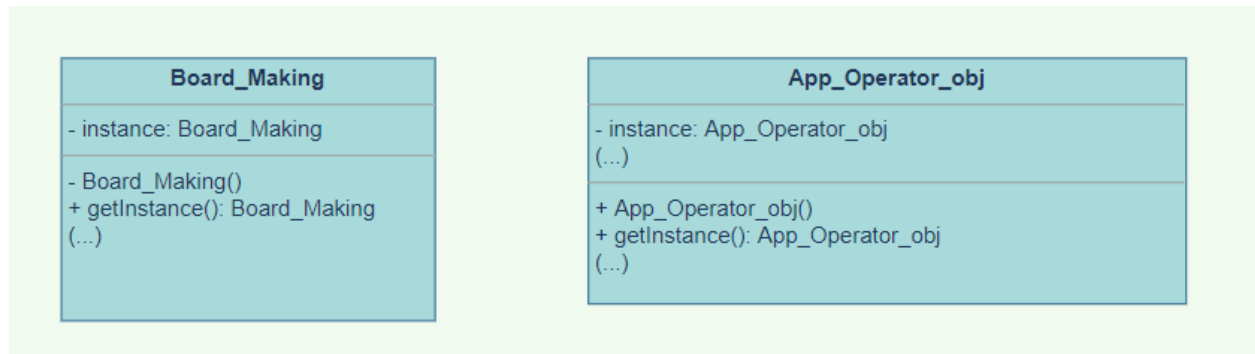
```

*Figure 33: Find all empty cells using BFS.*

## VII. Design patterns

### 1. Singleton Pattern

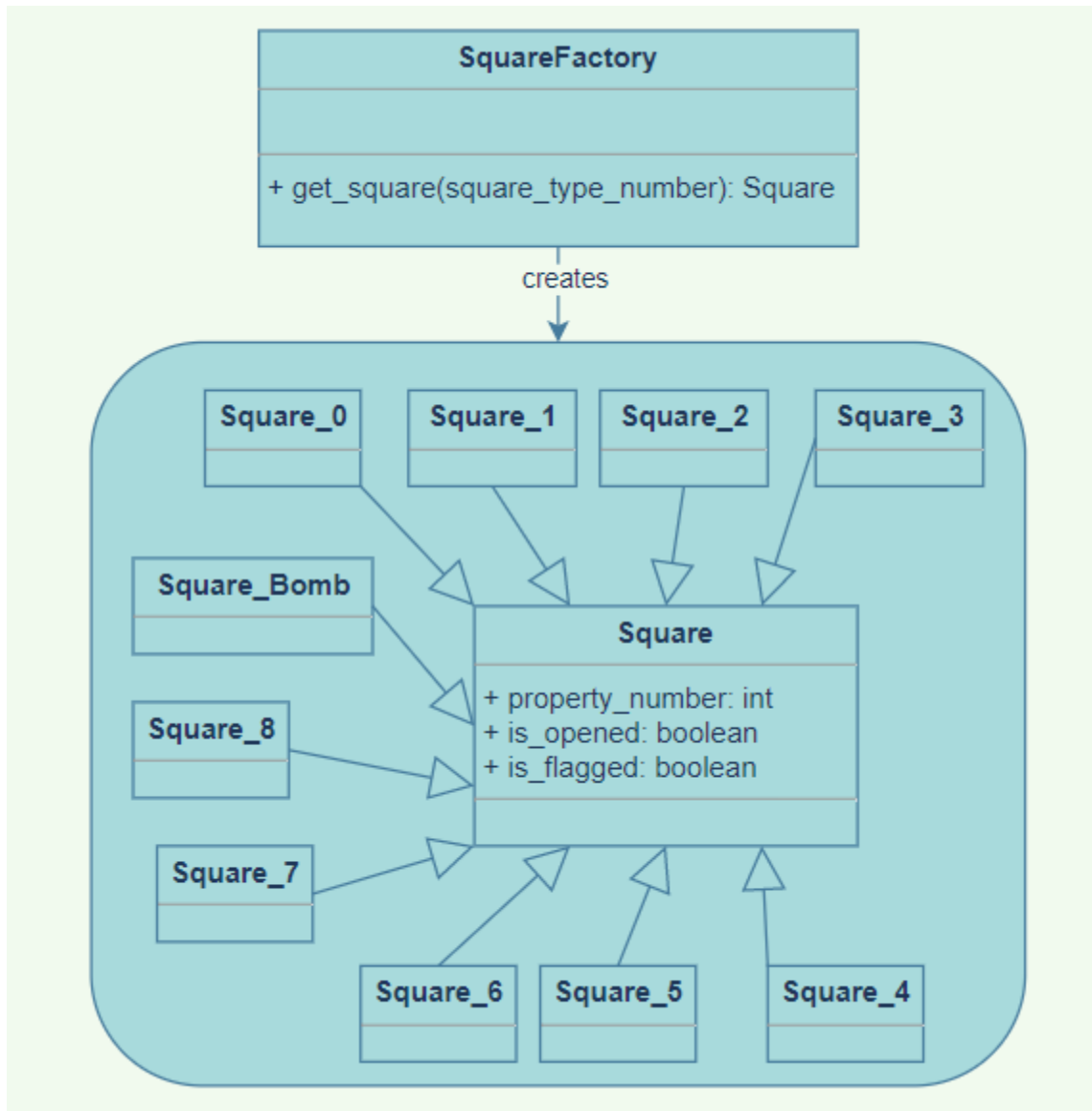
We have 2 classes that manage the entire system as creating the board, recording the state of the cells, or checking if the player win or lose, and are responsible for accessing and storing information. So we use Singleton Pattern for these 2 classes.



*Figure 34: Singleton Pattern*

## 2. Factory Pattern

We have many classes for different types of cells, for example cells with bombs inside, or cells with 7 bombs around. To facilitate creating objects for each cell, we need a class that can create 10 different types of objects using different parameters. To solve this problem, we use the Factory Method Pattern, and our SquareFactory class is used to create different objects.



*Figure 35: Factory Method Pattern*

## VIII. Conclusion

### 1. Experience

During the project, we have had the experience to:

- + Bring some web technologies into practice
- + Working as a team and evaluating team work performance

+ Implementing some algorithms (BFS, Queue, ...) from the DSA course.

## 2. Problems

We also found some difficulties during the project. The code base design was not appropriate which causes pain to read and continue implementing further features. This causes excessive team communication to explain what the code does in many specific parts, which is not time-effective.

## 3. Recommendations

From the difficulties arising in this project, we have drawn some solutions as follows:

+ We need to learn Model – View – Controller (MVC) Pattern to make the code more organized and cleaner. Therefore, it will make the code easier to read and maintain in the future.

+ We need to plan carefully before working on the project because with nothing in our mind, we really don't know what to do and how to assign task for members, so the project is very messy at the beginning. It was not until the middle of the project that we realized this and fixed it. Therefore, we need to plan carefully for the project as soon as possible.

## 4. Future improvements

It is wonderful for our team to create the API for creating the random positions for the bombs and all information of the board. With the API, we can expand the app to use on more platforms as window app or mobile app. Moreover, we will improve the code organization by using MVC or MVVM to make the code easier to maintain and expand more interesting features.

## IX. References

1. <https://www.toptal.com/nodejs/secure-rest-api-in-nodejs>
2. <https://www.codegrepper.com/code-examples/javascript/javascript+count+time>