Technical Report: Minesweeper Using Java Swing

COE102.2: B23 - Object Oriented Programming

Tristan O. Jadman
tristan.jadman@g.msuiit.edu.ph

Mark Andrey Acebu
markandrey.acebu@g.msuiit.edu.ph

Department of Electrical Engineering and Technology
College of Engineering
Mindanao State University - Iligan Institute of Technology
Iligan City, Philippines

I. Introduction

A. Overview of Minesweeper game:

Minesweeper is a classic single-player puzzle game that originated in the 1960s and gained popularity with the release of Microsoft Windows in the 1990s. The player's goal is to uncover all the cells except those containing mines. Each cell reveals a number indicating how many mines are adjacent. Using these clues, the player must carefully uncover the cells and avoid the mines to win the game.

B. Purpose of the report:

This report aims to explain how the Minesweeper game works and explain how it was implemented in code. In this report, the developers will discuss the game mechanics, such as revealing and flagging cells, and explain how the code handles these actions. Additionally, they will provide visualizations and screenshots of the game interface to help the audience better understand how to play Minesweeper.

II. Game Mechanics

A. Objective of the game

Minesweeper is a game that tests the player's thinking skills and requires strategic decision-making to win. The objective of the Minesweeper game is to uncover all the cells on the board that do not have mines. The challenge is to reveal each cell while avoiding the hidden mines. The objective is to clear the entire board without triggering any mines.

To succeed in Minesweeper, the player needs to think logically, use the given information of the cell that the player uncovers, and make smart moves to uncover all safe cells while flagging the dangerous mines. If the player clicks a cell that contains a mine, It's game over.

B. Game rules and gameplay

1. Revealing cells

- **findEmptyCell method**: Reveals neighboring empty cells recursively when an empty cell is clicked, stopping at numbered cells or mines.
- **mousePressed method in the MinesAdapter class**: Handles left mouse button clicks. If the clicked cell is eligible for revealing (not already revealed or flagged), it reveals the cell. If it's an empty cell, it reveals neighboring empty cells using the findEmptyCell method.

2. Flagging cells

- The **mousePressed method in the MinesAdapter class**: handles right mouse button clicks. It checks if the clicked cell can be flagged (not already revealed). If it can be flagged, it adds a flag to the cell. Subsequent right-clicks on a flagged cell remove the flag from the cell.

3. Winning and losing conditions

- **Winning Condition**: If all non-mine cells have been uncovered (uncover count becomes zero) and the game is still in progress, it means the player has won. The status is updated to "You won!".
- **Losing Condition**: If the game is not in progress, it means the game is over. The status is updated to "Game Over!" and a beep sound is played.

III. Implementation Overview

A. Programming language and framework used

The Minesweeper game is implemented using Java programming Language and it utilizes the graphical user interface or GUI toolkit provided by Java, **Java Swing**. Java Swing allowed the developers to create an interactive and visually appealing user interface for the game. They mainly used the **JFrame**, **JPanel**, and **JLabel**, which are Swing components, to build the game window, display game elements, and handle user interactions.

Java Swing is known for its **lightweight nature**, making it suitable for applications with simple to medium complexity, such as Minesweeper. It has a smaller footprint *(requires less memory and disk space compared to the JavaFX framework)*compared to JavaFX, making it suitable for applications with limited resources or where performance is a critical factor. Swing's simplicity also means that it has a **shallow learning curve** , making it accessible for developers new to GUI programming.

B. Game architecture and design choices

The Minesweeper game architecture employs a **modular and object-oriented design** approach to ensure code organization and maintainability. At the core of the design is the Board class, which manages the game's state, user interactions, and visual representation. This class incorporates an array-based data structure to represent the game grid and utilizes Java Swing for rendering the game board and user interface elements.

The design choices also embrace **event-driven programming**, where mouse events such as clicks and right-clicks are captured by event listeners and processed by the Board class. These events trigger actions such as uncovering cells or flagging them, resulting in updates to the game state and appropriate visual feedback.

C. Overview of major classes and their responsibilities

Board Class:

- Responsible for managing the game board, game state, and user interactions.
- Handles the generation of mines and the placement of numbers indicating adjacent mines.
- Manages the logic for uncovering cells, flagging cells, and determining win or loss conditions.
- Interacts with the GUI components to update the visual representation of the game board.
- Responsible for loading and storing the images used for the game's visual representation.
- Loads the images from the resource files and stores them in an array for easy access.
- Provides methods to retrieve the appropriate image based on the cell state.
- Renders the game board, including cells, numbers, flags, and mines, using the loaded images.
- Receives user input through mouse events and delegates them to the MinesAdapter class for processing.
- Displays the current game status, such as the number of mines left.
- Represents the graphical user interface (GUI) panel that displays the game board and controls.

MinesAdapter Class:

- Serves as a mouse event listener for the game board.
- Detects and processes user mouse clicks and right-clicks on the cells.
- Triggers corresponding actions, such as uncovering cells or flagging cells, based on the mouse events.
- Updates the game state and GUI components accordingly.

Minesweeper Class:

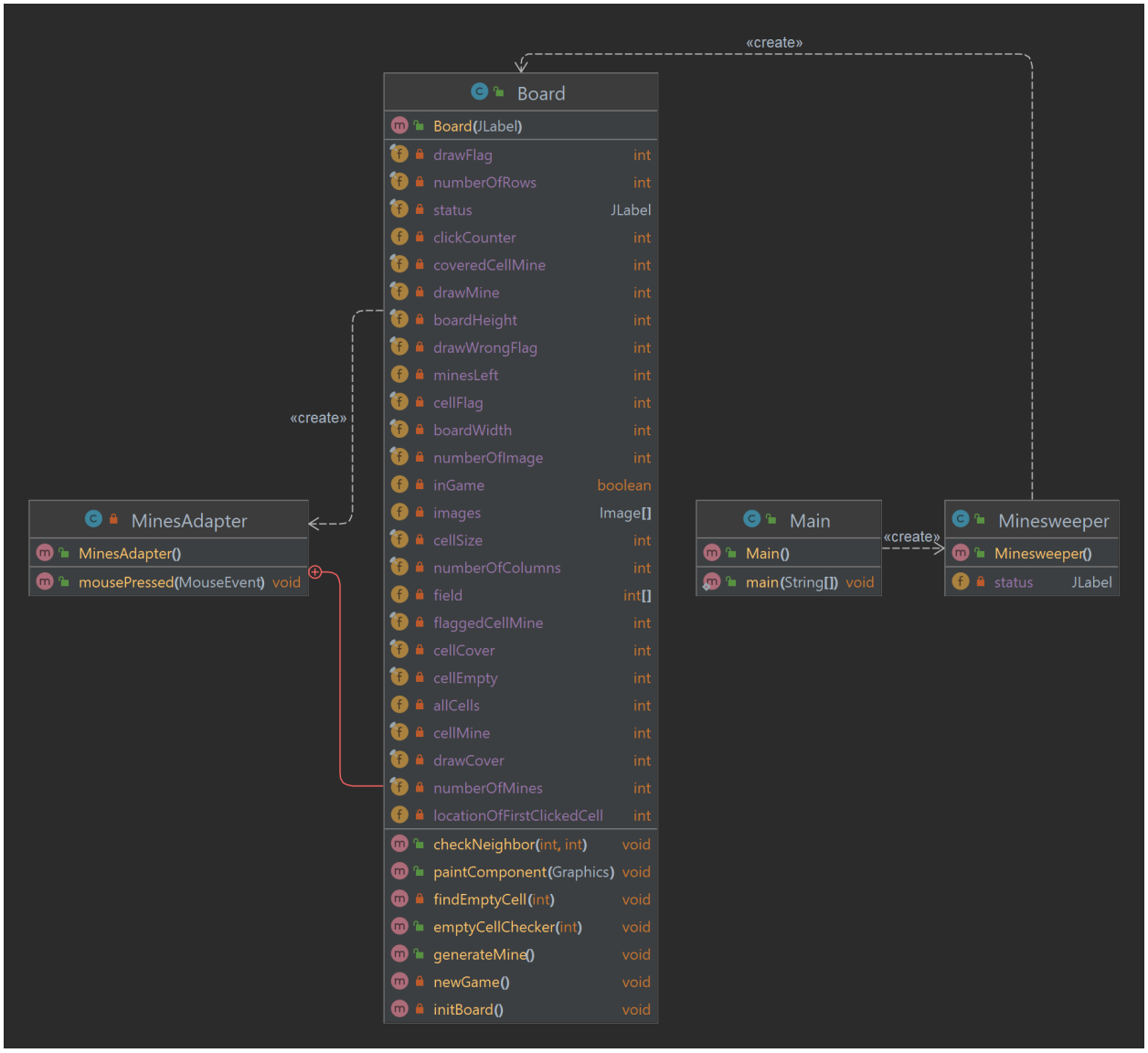
- Responsible for creating an instance of the Board class and setting the location of the status
- Sets the title on the title bar and changes the image icon of the window
- Sets the JFrame to exit on close

Main Class:

- Responsible for running the code.
- A main class that creates an instance of the Minesweeper class and sets the window as visible

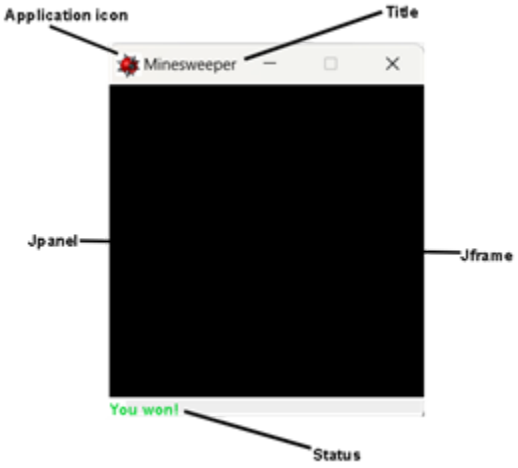
D. UML Class Diagram

To visualize the algorithm system, provided below is the UML of the Class that is used in the Minesweeper game.



IV. Game Implementation Details

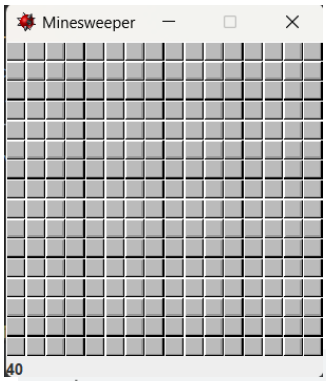
A. Game initialization



1. Setting up the Stage

Initialize the game window by setting up the window size based on the preferred size set inside the board. Adding the title of the game and setting up an icon that will be shown at the top left corner of the game window and at the taskbar or even on your desktop. More to it is calling methods that act as a condition to display the game window at the center of the screen. Also, terminating the algorithm once the game window is closed.

```java
public class Minesweeper extends JFrame {
    private JLabel status;

    public Minesweeper() {

        ImageIcon icon = new ImageIcon("src//resources//13.png");

        status = new JLabel("Status");
        add(status, BorderLayout.SOUTH);
        add(new Board(status));
        setResizable(false);
        pack();
        setTitle("Minesweeper");
        setIconImage(icon.getImage());
        setLocationRelativeTo(null);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
```



2. Board size and mine placement

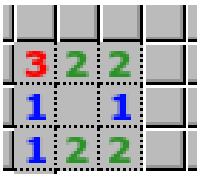
When a new game starts, each cell is represented by a default value that will be altered as the game is in progress, and that value corresponds to the image that the paintComponent will generate in that location of the cell every mouse clicks.

Java

```
private void initBoard(){
       setPreferredSize(new Dimension(boardWidth, boardHeight)); //Set UI size
//existing code
       }
  public void generateMine() {
       var random = new Random(); //New object of Random class
        int i = 0;
        while (i < numberOfMines) {
            int minesPosition = 0;
            if(clickCounter == 1) {
       //Generate random positions for the mines
                minesPosition = (int) (allCells * random.nextDouble());
            }
       //Never repeat mines position and except the location of the first
clicked cell
            if ((minesPosition < allCells) && (field[minesPosition] !=
coveredCellMine) && minesPosition != locationOfFirstClickedCell) {
                int current_col = minesPosition % numberOfColumns;

              field[minesPosition] = coveredCellMine;
       //Check the neighbors of the planted mine
                checkNeighbor(current_col, minesPosition);
                i++;

            }
        }
    }
```

B. Game Logic

1. Check neighbor



This checks the neighboring cell of the clicked cell and starts from the upper left moving down which is the left of the clicked cell and then the bottom left cell. It will then move to the upper cell of the clicked cell and then the bottom cell. From there, it moves to the upper right cell and then the right cell, and lastly the bottom right cell.

Java

```
 private void findEmptyCell(int clickedCell) {

        int current_col = clickedCell % numberOfColumns;    //Column of the
first clicked cell
        int cell;

        if (current_col > 0) {    //Checks if the left-side neighbors exist
            cell = clickedCell - numberOfColumns - 1; //Top-left adjacent cell
            if (cell >= 0) {      //If cell location is part of the game
                emptyCellChecker(cell); //Checks cell if empty
            }

            cell = clickedCell - 1;     //Left-side adjacent cell
            if (cell >= 0) {            //If cell location is part of the game
                emptyCellChecker(cell); //Checks cell if empty
            }

            cell = clickedCell + numberOfColumns - 1; //Bottom-left adjacent
cell
            if (cell < allCells) {      //If cell location is part of the game
                emptyCellChecker(cell); //Checks cell if empty
            }
        }

        cell = clickedCell - numberOfColumns; //Top adjacent cell
        if (cell >= 0) {                //If cell location is part of the game
            emptyCellChecker(cell);     //Checks cell if empty
        }

        cell = clickedCell + numberOfColumns;  //Bottom adjacent cell
        if (cell < allCells) {          //If cell location is part of the game
            emptyCellChecker(cell);     //Checks cell if empty
        }
```

```
        if (current_col < (numberOfColumns - 1)) {    //Checks if right-side
neighbors exist
            cell = clickedCell - numberOfColumns + 1;       //Top-right
adjacent cell
            if (cell >= 0) {            //If cell location is part of the game
                emptyCellChecker(cell);       //Checks cell if empty
            }
            cell = clickedCell + 1;     //Right adjacent cell
            if (cell < allCells) {      //If cell location is part of the game
                emptyCellChecker(cell); //Checks cell if empty
            }
              cell = clickedCell + numberOfColumns + 1;     //Bottom-right
       adjacent cell
            if (cell < allCells) {      //If cell location is part of the game
                emptyCellChecker(cell);       //Checks cell if empty
            }
        }

    }
```

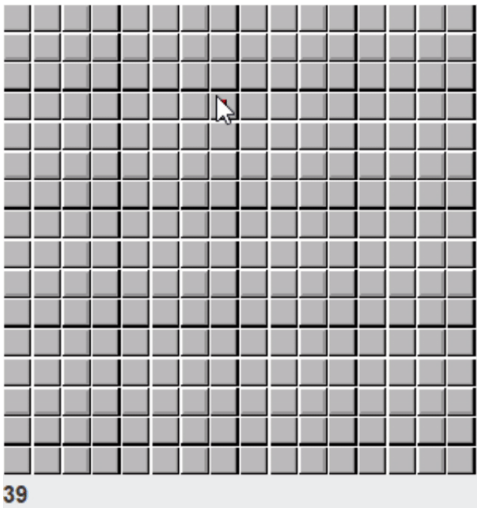
2. Uncover empty cells

This uncovers the cell if the player hasn't clicked on that cell. It will then determine if the cell is empty or if it has a value *(1,2,3,4,5,6,7,8)* . If it's empty it should return the checked cell and will re-process the findEmptycell having the checked cell as the new cell to process from top left to bottom right. This will run up until the code cannot find an empty cell.

Java

```
public void emptyCellChecker(int cell) {
        if (field[cell] > cellMine) {   //If cell is not a mine
            field[cell] -= cellCover;   //Uncover the cell
            if (field[cell] == cellEmpty) {    //If cell is empty or equal to 0
                findEmptyCell(cell);    //Indirect recursive call
            }
        }
```

C. Event handling



1. Mouse clicks and their effects

This will get the location of where the player clicks and process it depending on what button the player presses. The right click mainly is for flagging and unflagging a cell while the left click is the one that uncovers the cell. Over time, the game will re-process to check the condition of the game and update the game status accordingly.

Get the current location of where the player clicked the mouse and run a condition to re-process the game.

Java

```
@Override
 public void mousePressed(MouseEvent e) {
      int x = e.getX();   //x-coordinate of the mouse
     int y = e.getY();    //y-coordinate of the mouse

     int cCol = x / cellSize;    //Current column of the mouse
     int cRow = y / cellSize;    //Current row of the mouse

     boolean doRepaint = false;  //repaint() checker

     if (!inGame) {       //If game is not in progress
         newGame();       //Starts new game
         repaint();       //Refreshes the painting of the Board
     }
```

This handles the condition for the right click or Button 3 as indicated here using the Mouseevent.The right-click toggles the flagger functionality for the suspected mine cell and removes the flag if right-click is used on the same cell twice. This ensures that no cell is flagged twice. However, this functionality is limited depending on the minesLeft status. If the mines left status is zero, the player can no longer add a new flag to a cell.

Java

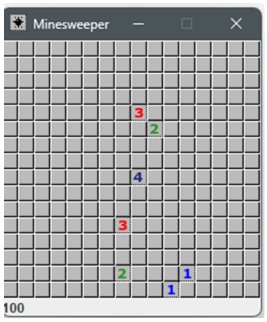
```
private class MinesAdapter extends MouseAdapter {

        @Override
        public void mousePressed(MouseEvent e) {
//existing code
            if ((x < numberOfColumns * cellSize) && (y < numberOfRows *
cellSize)) {  //If mouse event is within GUI

                if (e.getButton() == MouseEvent.BUTTON3) { //RIGHT-CLICK is a
flagger functionality
       //If clicked cell is not an uncovered mine
                    if (field[(cRow * numberOfColumns) + cCol] > cellMine) {

                        doRepaint = true; // enable repaint

                        if (field[(cRow * numberOfColumns) + cCol] <=
coveredCellMine) {  //If clicked cell is not yet flagged
             //Enable flagger if minesLeft status is not yet 0
                            if (minesLeft > 0) {
                                field[(cRow * numberOfColumns) + cCol] +=
cellFlag;    //Flags the cell
                                minesLeft--;   //Decrease the minesLeft status
                                String msg = Integer.toString(minesLeft);
                                status.setText(msg); // Updates status
                            } else { //Disable flagger if minesLeft is zero
                                status.setText("No marks left");
                            }
                        } else {  //If cell is right-clicked twice, remove flag
and increment minesLeft status

                            field[(cRow * numberOfColumns) + cCol] -= cellFlag;
                            minesLeft++;
                            String msg = Integer.toString(minesLeft);
                            status.setText(msg);
                        }
                    }
                }
```

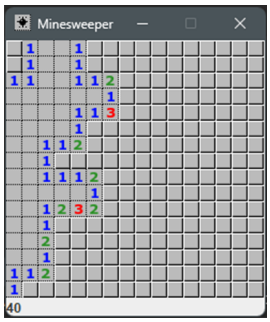
This checks all the left click actions. The left-click event mainly enables the uncovering functionality of the game. To be able to make the game safe on its first move, the click counter is implemented to generate the mines only after the player first clicks on a cell and saves its location. This location will then become an exception to the possible positions of the mine in the mines generation function, ensuring that the first click cell of a new game will not be a mine cell, eliminating the possibility of the player losing on his first move. Certain conditions like disallowing flags to be left clicked. Otherwise, a cell will uncover if it was covered and is not flagged. It will then push the cell that is empty through the findEmptyCell function to check game logic.

```
Java
private class MinesAdapter extends MouseAdapter {

        @Override
        public void mousePressed(MouseEvent e) {
//existing code

else { //LEFT-CLICK uncovers the cell
                    clickCounter++;
                    if(clickCounter == 1 && inGame) { //Generates mine after
the first click of a new game
                        locationOfFirstClickedCell = (cRow * numberOfColumns) +
cCol;  //Saves the location of the first clicked location
                        generateMine();
                    }

                    if (field[(cRow * numberOfColumns) + cCol] >
coveredCellMine) { //If clicked cell is flagged, no changes
                        return;
                    }

                    if ((field[(cRow * numberOfColumns) + cCol] > cellMine)
                            && (field[(cRow * numberOfColumns) + cCol] <
flaggedCellMine)) { //If clicked cell is not yet an uncovered and not a flagged
cell, uncover the cell
                        field[(cRow * numberOfColumns) + cCol] -= cellCover;
//remove cover
                        doRepaint = true; //Enable repaints
                        if (field[(cRow * numberOfColumns) + cCol] == cellMine)
{            //If uncovered cell is a cell mine
```

```
                            inGame = false;    //Game's over
                        }
                        if (field[(cRow * numberOfColumns) + cCol] ==
cellEmpty) {  //If cell is empty
                            findEmptyCell((cRow * numberOfColumns) + cCol);
//Find all neighboring empty cells recursively
                        }
                    }
                }
```

D. Rendering the game board





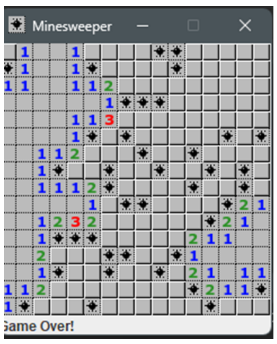
1. Displaying cells and their states

Presenting the individual game elements, along with their current state. This involves rendering the cells on the screen and updating their appearance to reflect changes in their states, such as uncovering clicked cells, indicating an empty cell or cell containing a number, or showcasing the numbers of mines left based on the number of flagged cells.

Java

```
 @Override
    public void paintComponent(Graphics g) {

        int uncover = 0;

        for (int i = 0; i < numberOfRows; i++) { //Up to rows (horizontally)
            for (int j = 0; j < numberOfColumns; j++) { //Up to columns
(vertically)

                int cell = field[(i * numberOfColumns) + j];

                if (inGame && cell == cellMine) { //In game and clicked a mine
                    inGame = false; //Stops the game
                }

                if (!inGame) {
                    if (cell == coveredCellMine) {
```

```
                        cell = drawMine; //Reveals all mines after inGame is
false
                    } else if (cell == flaggedCellMine) {
                        cell = drawFlag; //Show if flag is correctly placed
                    } else if (cell > coveredCellMine) {
                        cell = drawWrongFlag; //Show if flag is incorrectly
placed
                    } else if (cell > cellMine) {
                        cell = drawCover; //Uncovered empty cells remains
uncovered
                    }

                } else { //Game in progress

                    if (cell > coveredCellMine) { //Flagged all possible boxes
(EMPTY CELL -> TO MINES)
                        cell = drawFlag;
                    } else if (cell > cellMine) {
                        cell = drawCover;
                        uncover++;
                    }
                }
               g.drawImage(images[cell], (j * cellSize), (i * cellSize), this);
//Draw cell image based on the value it holds
            }
        }
```



2. Handling game states (in progress, win, loss)

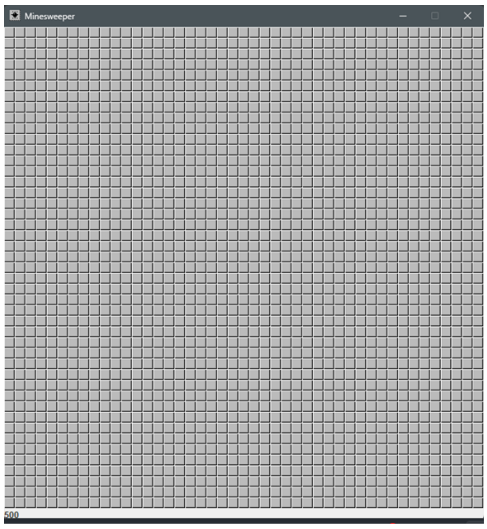
To actively update the progress of the game, the developers have set up the status variable which is a JLabel variable. This code would then call setText from the JLabel component. It runs through the condition for the "You won", and "Game over" status of the game.

```Java
 @Override
    public void paintComponent(Graphics g) {
    //existing code
```

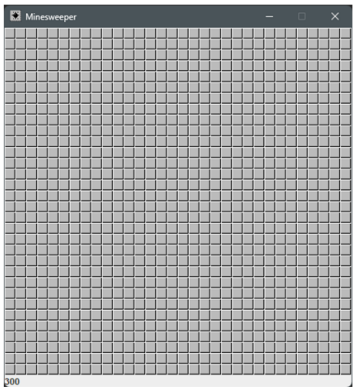
```
 if (uncover == 0 && inGame) {    //If all cells except the suspected mine cells
are uncovered and game is still in progress
            inGame = false;       //Stop the game
            status.setText("You won!"); //Update status
        } else if (!inGame) {           //If game's over or player loses
            status.setText("Game Over!");      //Update status
            Toolkit.getDefaultToolkit().beep();
        }
```

V. Visualization of Application

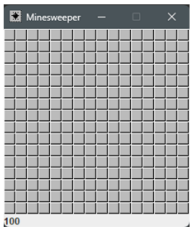
A. Screenshots or illustrations of the Minesweeper game



45x45
500 MINES

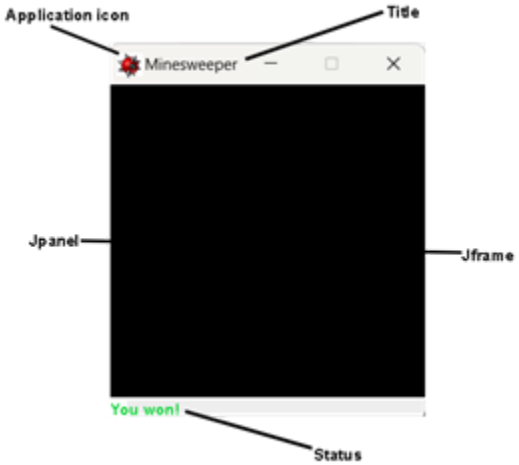


32x32
300 MINES



16x16
100 MINES

B. Parts of the Game window



VI. Conclusion

A. Summary of the game mechanics and implementation

To conclude, in order to win the game, the player must uncover all the cells that do not contain a mine by using the given information of the uncovered cells and using the flagging feature to put flags and restrict the player to click those flagged cells. Otherwise, when the player uncovers a cell with a mine, the player loses.

The utilization of the components brought by the Swing with the integration of Java AWT (Abstract Window Toolkit) framework, has made it possible for the developers to successfully produce the game.

Swing's lightweight nature and component-based architecture have made it easier for the developers to implement the logic of the game and manage and optimize the game structure. The rich library of components of Swing following an event-driven programming model, allowed them to customize the GUI to provide a visually appealing graphical user interface (GUI) and handles user interaction through the event listener. Upon leveraging the components effectively, the logic of the game is implemented as code in such a way that it surpasses readability, scalability, efficiency, and testability.

To conclude, the project has opened a gateway of opportunity for the developers to experience the development of a game through the application of their knowledge in Object Oriented Programming. The development of the game Minesweeper has made rewarding educational benefits to the developers. This project allowed them to learn, hone, showcase, and apply their talents, skills, and learnings in the Object Oriented Programming course and deepen their understanding of GUI development. It stands as a testament to their dedication and serves as a stepping stone for their future endeavors in software development especially in game development.

B. Challenges faced during the development

Board formula: Initially, the developers tried to make a 2D array to generate the dimensions of the board. This process led them to manually generate boxes independently from the board dimensions.

Debugging this required them to further browse the internet for functions that would allow them to use the board dimension and generate the cells accordingly.

The dimensions of the window now are dependent on the rows and columns that they set in the game.

Mine Generator Bug: Initially, the mines were being generated as soon as the game started, leading to an incorrect setup of the game board.

Debugging this required the developers to implement a click counter. The click counter ensured that mines would only be generated once the player clicked on a cell for the first time. This fix resolved the bug and ensured that the mines were appropriately placed in the game.

By implementing the click counter, they were able to control the timing of mine generation and ensure that it occurred when the player initiated the game.

Neighbor checker loop: Before stumbling into our current algorithm the developers first thought out of only revealing one cell at a time which was easier to do but would take longer time to play.

Overcoming this challenge required careful debugging, testing, and refinement of the neighbor checker algorithm. The developers then tried out to loop the empty cell checker around each box and applied an indirect recursion to update the clicked cell and this would run through up until there is no empty cell detected.

C. Future enhancements or improvements

- **Timer**: One thing that the developers didn't add from the original game to their version of the game is a timer. This is due to the fact that it is useless without a database that could store the time the player finishes the game.

- **High Score Database**: Implementing a database would require more time to build and as new game developers, they are still starting and lack the complete knowledge to implement a database even if we know the logic behind it.

- **Better UI**: This is an optional enhancement since a lot of software has been developing to a much better UI that mainly follows a simple and clean look. It was not implemented in the game in order to reminisce the nostalgic feeling of playing a classic game.

- **Animations:** Animation plays a key role in making a game more interactive and enjoyable. This enhancement is one of the great milestones for this project.

- **Sound Effect**: This is one of the things that makes a game nostalgic remembering the music and sound effects of every move the player does. Adding it to the game would then make the game more engaging and fun to play.

- **Code Structure**: The developers cover the idea from the JavaFX about the stage and scene and that is mainly why the code was structured as if we are coding the stage and then separating the scene. For future development, implementing POJO and other coding methods could further enhance the structure of our code and improve readability.

VII. References

A. External resources used or referred to

Minesweeper (video game) - Wikipedia
History of Minesweeper: 20 Things To Know (Origins, Microsoft,...) - Gamesver
When to use JFrame, JLabel, and JPanel - Stack Overflow
java.awt.Image from File - Stack Overflow

B. Credits for any assets or libraries utilized

Import Statement	Description
import java.awt.BorderLayout;	Provides layout manager for positioning components within the game window.
import javax.swing.JFrame;	Enables the creation and management of the game window as a JFrame.
import javax.swing.JLabel;	Represents a simple text label used to display information in the game UI.
import java.awt.event.MouseAdapter;	Allows the implementation of custom mouse event handling for user interactions.
import java.awt.event.MouseEvent;	Represents mouse events that occur within the game window.
import java.util.Random;	Provides functionality for generating random numbers for mine placement.
import javax.swing.ImageIcon;	Represents an image icon that can be used for graphics within the game.
import javax.swing.JPanel;	Provides a base class for creating custom panels within the game UI.
import java.awt.Dimension;	Represents the dimensions of GUI components, used for setting panel size.
import java.awt.Graphics;	Enables graphics rendering on the game board panel.
import java.awt.Image;	Represents an image resource used for cell icons and graphics.

***Note**: The game was developed by both developers Tristan Jadman and Mark Andrey Acebu. The delegation of tasks cannot be determined as there are times when we work together in the same place. For further information, Tristan Jadman is the one who coded the paintComponent while Mark Andrey Acebu coded the MouseEvents but it is still a collaboration where we both brainstorm in coming up with how the game works and what methods we should implement before we start coding.*
https://github.com/Andreeeyp/OOP-Project.git