Design Manual

Introduction

Our application is a basic Minesweeper game. We took inspiration for the styling from the google Minesweeper rather than the classic game. The application as a whole is a javaFX application, which allows for us to show information graphically, and allow for user input through the mouse. From a technical standpoint, each square in the game can be viewed as two separate components, a rectangle and a label. Each of these serves their own purpose, with the rectangle being the color, and the label showing the information. When it comes to the logic behind the game process, the most notable logic is behind each user click. When a user makes a move on the board, there is usually recursion involved in order to uncover empty cells, and show neighboring non-empty cells.

User stories

1.) Edwin Watkins

As Edwin Watkins the Achiever player, I want to play a game that is challenging and complex so that I can defeat other players or out-play the computer.

Quote: “I am a winner in all aspects of life”

Narrative: Edwin plays games to win. He enjoys single player games where he can compete against the computer or multiplayer games where he can compete with other players. He is very competitive.

2.) Elizabeth Wood

As Elizabeth Wood the Social Player, I want to play a game that is interactive and has a lot of detail so that I have fun playing the game.

Quote: “I really enjoy playing games that are interactive and vibrant, it makes it so fun to play!”

Narrative: Elizabeth enjoys games that are interactive and have a lot going on on the screen. She likes one player games that she can play wherever, whenever

3.) Elle Aarvik

As Elle Aarvik the Fun player, I want to play a single player game just for fun so that I can enjoy the game and not worry about competing with someone else.

Quote: “I’m just here to have fun”

Narrative: Elle enjoys playing games with no goal other than to have fun. She prefers single player games where she can choose her own paths and not worry about competing with another person or winning.

4.) Oliwier Levang

As Oliwier Levang the Data Analyst, I want to understand how the game and visuals were created so that I can create similar work.

Quote: “I love working with data”

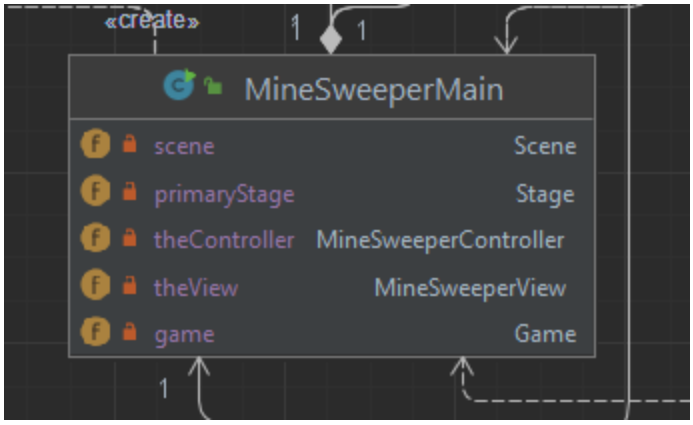
Narrative: Oliwier enjoys the process of cleaning data and making it usable for data visualizations. He believes that when it comes to creating data visualizations that data cleaning is one of the most important parts.

Object Oriented Design - All Classes

The Main Class - MineSweeperMain

The highest level class in our program is MineSweeperMain. There are only a couple things to note in this class. Mainly, this class sets up the JavaFX application and creates the instances of MineSweeperView, MineSweeperController, and Game. But it also contains the resetGame() function which provides the functionality to reset the game when a new difficulty is chosen, or the user loses their current game by clicking a bomb.

```
public void resetGame(DIFFICULTY difficulty){
    try {
        Game tempGame = new Game(difficulty);
        MineSweeperView tempView = new MineSweeperView(tempGame);
        MineSweeperController tempController = new MineSweeperController(tempView, tempGame,  main: this);

        this.game = tempGame;
        this.theView = tempView;
        this.theController = tempController;

        start(primaryStage);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```



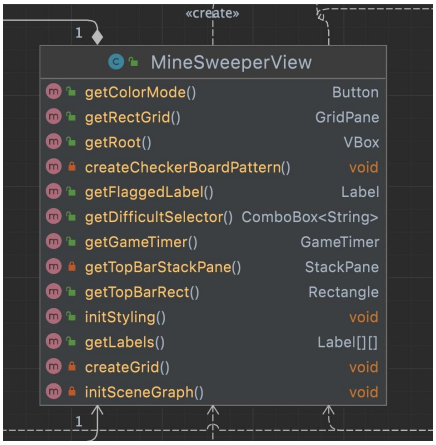
Admittedly, as will be obvious soon, our UML diagram is extremely messy. Here it can be seen that there are many dependencies on MineSweeperMain.

The View Class - MineSweeperView

As is consistent with most JavaFX applications, our view class handles all components that will be visible to the user. This also includes functionality and any logic needed to style these components.

At the highest level, our application's view is broken into two main parts. An HBox - topBar, and a GridPane - rectGrid. topBar encapsulates all the information regarding the state of the game, along with game options for the user to interact with. There is a drop down menu where a difficulty can be selected, along with a button that allows the user to switch between light mode and dark mode.

In the GridPane, every element is a StackPane which holds a Rectangle and a Label. The rectangle is only there to provide a colored background, while each label holds necessary visible information for that cell. This would include numbers to represent the number of neighboring bombs, or the letter 'F' which tells the user that the cell is flagged.



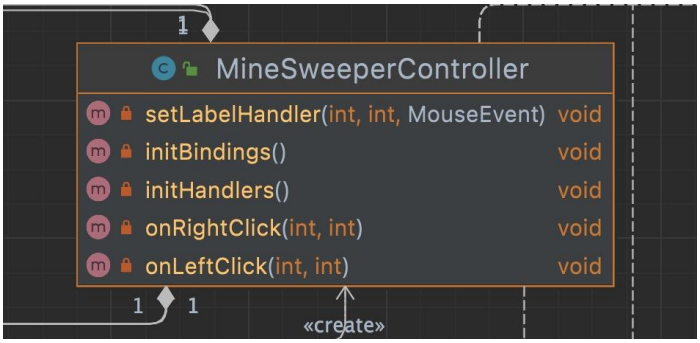

```
private void createGrid() {
    StackPane sPane;
    Rectangle currRect;
    for (int r = 0; r < game.getRowCount(); r++){
        for (int c = 0; c < game.getColCount(); c++){
            currRect = new Rectangle(gridSize, gridSize);
            labels[r][c] = new Label( text: " ");

            sPane = new StackPane(currRect, labels[r][c]);
            rectGrid.add(sPane,c,r, colspan: 1, rowspan: 1);
        }
    }
}
```

Here, the UML class for View can be seen. Along with this, the logic behind creating the background grid of cells is shown.

The Controller Class - MineSweeperController

As in the majority of JavaFX applications, the controller class handles all bindings and user inputs. Also, this class handles the major of the game events, such as a mouse left-click or a mouse-right click.

At the highest level, our controller initializes the bindings, which binds the labels text property(see View Class) to each cells display String, the TopBar width to the Width of the Rectangular Grid, and the Flagged Label to correctly display how many flags the user has remaining.




```
private void onLeftClick(int finalR, int finalC){
    if (cells[finalR][finalC].isFlagged()) return;

    if (!game.playerMove(finalR, finalC,  flagging: false, !hasClicked)){
        theView.getGameTimer().stop();
        endGamePopUp = new EndGamePopUp(main,  message: "GameOver!", currentDifficulty);
        endGamePopUp.initStyle(isDark);
        endGamePopUp.show();
    }

    for (Cell cell : game.getVisitedCells()) {

        if (cell.isDarkTile()) {
            labels[cell.getRow()][cell.getColumn()].getStyleClass().add("exploredTile");
        }else{
            labels[cell.getRow()][cell.getColumn()].getStyleClass().add("exploredTileLight")
        }
    }
}
```

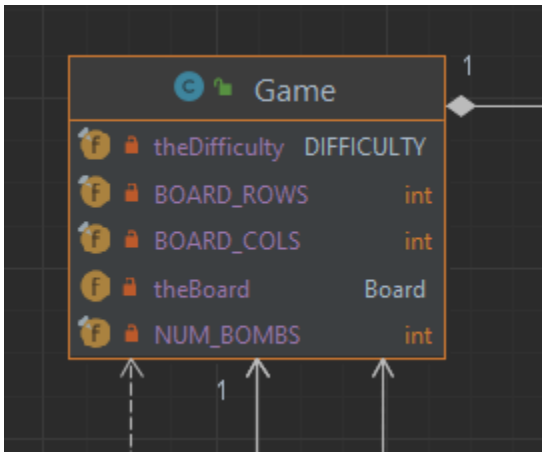
Here, the UML class for Controller can be seen. Along with this is how a left-click is handled.

The EndGamePopUp Class

The EndGamePopUp class is a class that handles endGame behavior. It can be created with a game Winner or game Loser message and allows for the user to play again, initiating a new game, or exit resulting in the program to be closed. Furthermore, this class is styled to match the current color mode of the application.

The Game Class

The Game class does not involve as much logic as some of the other classes. The main purpose of the class is to create the Board instance as well as interact with the Board instance when user input is provided from the Controller class. Additionally, early on in the development process the Game class provided functionality for printing the board in an ascii format in the console. This allowed us to create a quick prototype of the game without having to worry about JavaFX just yet.This method is no longer used, although it was an important part of the development process.



The Board Class

The Board class is our most logic intensive class in this program. It handles everything from recursing through the cells when an empty cell is clicked, to initializing the board and placing bombs.

This might not be expected, but the cells in the board are not instantiated when a Board instance is created. Rather, this does not occur until the user makes the first move of the game. This can be seen in handleCell(). The purpose of this is to guarantee that the user's first move will not be on top of a bomb.

```
public int handleCell(int r, int c, boolean firstMove){
    if (firstMove)
        initCells(r, c);
    return handleCellHelper(r,c,visited);
}
```

If a move is made, and it is not the first move of the game, the location of the move will be passed to handleCellHelper(). This function has a few different purposes. If the cell that was clicked is a bomb, then the game will end by returning 1. If the cell is not empty, then the

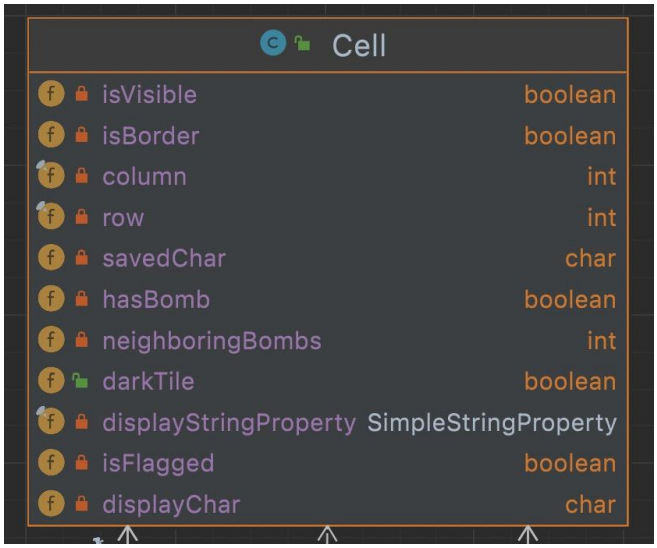
number of neighbors will be shown. Lastly, if the cell is empty, then the function will recurse through the area, making the cells visible, until non-empty cells are found.

The initiation of the Cells involves looping through the entire expected size of the Board, instantiating each Cell and setting its display value to " ", among some other things. However the placement of the bombs is a bit more complicated. When initBombs is called, a while loop is started which will continue to loop until the number of bombs placed has reached the desired amount. This while loop creates a random point on the board, then only places a bomb if the number of neighboring bombs at that point is less than 4, and the distance to the player's starting point is greater than 2. This guarantees that the spread of the bombs is fairly uniform, and it also guarantees that the player gets a decent clear area in which to begin the game. Lastly, when a bomb is placed, initNeighbors() is called which will add one to the neighborCount of each neighboring cell around that bomb.

```
private void initBombs(int startR, int startC){
    int bombCount = 0;
    while (bombCount < numBombs){
        int r = (int)(Math.random() * (cells.length - 2)) + 1;
        int c = (int)(Math.random() * (cells[r].length - 2)) + 1;
        if (!cells[r][c].isHasBomb() && countNeighboringBombs(r,c) < 4 && distanceTo(cells[startR][startC],cells[r][c]) > 2){
            cells[r][c].setHasBomb(true);
            initNeighbors(r,c);
            bombCount++;
        }
    }
}
```

The Cell Class

The cell class is the lowest class of this implementation. The purpose of this class is to hold the information of an individual cell. Each cell has very different characteristics, which is why this class contains many different variables. This class was added onto through the development process as we found more information that each cell would need to contain. For example, we were unsure of how to keep track of whether or not each cell was styled dark or not. The advantage of having access to this is helpful for the unflagging game mechanic, since the cell needs to be colored back to the original color, allowing for the game board to appear aesthetically pleasing to the user.



The GameTimer class

The GameTimer class provides the functionality behind the timer shown in the topBar of the application. When the GameTimer is started, it starts a javafx Timeline. This, similarly to a TimerTask, allows a task to be run every x amount of time. In this case, every second a seconds counter is incremented, and the value of the label is also incremented. And since GameTimer extends the Label class, it is able to be directly added as a component to the View, making things a lot simpler.