Minesweeper Software Architecture

September 2025

Introduction and Goals

This project is a simplified version of Minesweeper designed to demonstrate the core functionality of the game. The goal is to provide users with a fun and intuitive experience, ensuring smooth gameplay from start to finish. This game has a nautical theme to enhance the experience.

Mines are placed randomly after the first tile click. Each tile on the board communicates its state clearly: whether it has no nearby mines, the number of adjacent mines, or if it remains unrevealed. Tiles are uncovered using a flood-fill pattern initiated by the user's click. Players can also place buoys to mark suspected mine locations.

Both the backend and frontend are implemented in Python.

Requirements Overview

The requirements for this Minesweeper implementation are as follows. The game board is a 10 by 10 grid with labeled rows and columns. At the start of the game, the user selects the number of mines to be placed on the board; if no selection is made, the default is set to 10 mines. Mines placement occurs after the first user click to ensure fairness. Additionally, the first clicked tile and the surrounding 3×3 area are guaranteed to be free of mines.

During gameplay, users can click on tiles to reveal their state. Each revealed tile indicates whether it is safe and, if applicable, the number of adjacent mines. Tile revelation follows an expanding pattern that continues outward until it encounters a mine, the edge of the board, or a tile with a nonzero adjacent mine count. Users may also place buoys on unrevealed tiles to mark suspected mine locations. The game continues until the user either clicks on a mine, resulting in a loss, or successfully reveals all non-mine tiles, resulting in a win.

Technical Context

This program follows a traditional two-tier architecture consisting of a backend and a frontend. The backend serves as the core logic layer, maintaining all game data and providing functionality like an API by manipulating and updating the frontend. The frontend, in turn, is responsible for presenting the game to the user. One exception to this separation of concerns is the Cell class, which not only stores state data but also determines the

image displayed for each cell based on its state (e.g., whether it contains a mine, is flagged, or has been revealed).

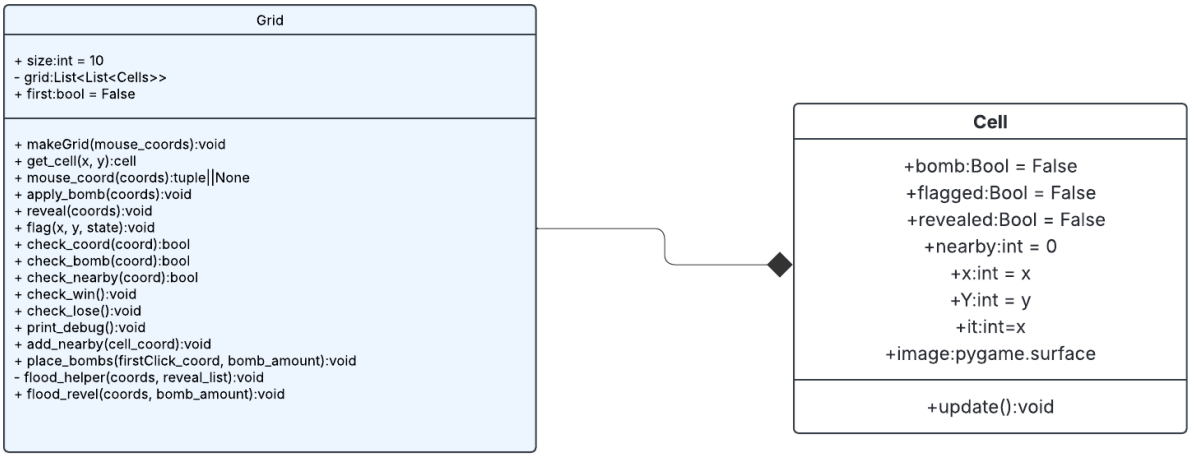
The Backend

The backend is composed of two primary classes: Cell and Grid. The Cell class should not be accessed directly by the front end; instead, it provides the necessary structure and state management to support the Grid class. The Grid class manages the overall game state and is called after each user action. Its core data structure is a two-dimensional list of Cell objects, and it contains the algorithms responsible for updating cell attributes, checking board conditions, and determining win/loss states.

The Frontend

The front end is implemented in Python using the PyGame library. Most frontend logic resides in main.py, except for the display behavior tied to the Cell class. The program generally follows a consistent flow: components are initialized at the top of the file, then referenced within the main game loop. User interactions are handled by two key conditional structures. The first processes mouse clicks events, which reveal cells, place buoys, or initiate a new game. The second determines which screen to render—the start page, game board, win screen, or game over screen—using global Boolean variables to track the current state of the game.

(The UML class diagrams shown below are to give context on how the two classes relate to each other)



Runtime View