Minesweeper - System Architecture Document

Team 28

The University of Kansas

EECS 581 - Software Engineering II

Nabeel Ahmad, Aniketh Aatipamula, Omar Mohammed,

Shero Baig, Humza Qureshi, Yaeesh Mukadam

System Architecture Document

Document Revision History

Date	Version	Comment	Author
08/26/2025	1.0	File is created, Outline is built for readability and future additions/revisions,	Nabeel Ahmad
09/04/2025	1.1	Added relevant information for Board Structure	Nabeel Ahmad
09/14/2025	1.2	Complete Components, Update Table of Contents,	Nabeel Ahmad, Omar Mohammed
09/19/2025	1.3	Add Relevant Diagrams, Complete Data Flow, Complete Key Data Structures, Complete Assumptions	Nabeel Ahmad, Omar Mohammed

Table of Contents

System Architecture Document

Section 1 - Purpose

The purpose of this project is to develop a single-player puzzle game known as Minesweeper. The game consists of a 10×10 grid with a random number of mines accounting for less than 20% of the grid squares. The grid has columns labeled A-J and rows numbered 1-10. Players uncover a cell by clicking on the cell. When a player uncovers a mine, the game ends. However, uncovering a mine reveals a number from 0-8 indicating adjacent mines. Cells with zero adjacent mines trigger recursive uncovering of adjacent cells. Players can toggle flags on covered cells to mark suspected mines. We implement this in the browser using a Next.js framework utilizing TypeScript and CSS. This allows for simpler Graphical User Interface (GUI) operability for the user and enables portability across various platforms.

Section 2 - Components

2.1 Board Manager

This component manages the 10×10 grid and tracks cell states. Cell states can be either covered, flagged, uncovered, or a mine).

Key responsibilities:

- Manage 10×10 grid
- Track each cell state for:
    - Covered (hidden, default state)
    - Flagged (player suspects a mine)
    - Uncovered (revealed, safe cell)
    - Mine (dangerous cell)

2.2 Game Logic

This component enforces Minesweeper rules. It processes uncovering, flagging, and recursive revealing of adjacent zero-valued cells. It detects game-ending conditions (loss when a mine is uncovered, victory when all safe cells are revealed) and communicates updated status to the User Interface. Upon board initialization, the Game Logic component ensures the first-clicked cell is guaranteed mine free.

Key responsibilities:

- Validate moves (ignore uncovering flagged cells)
- Count adjacent mines (0-8) for each revealed cell
- Handle recursive flood-reveal when zero mines are adjacent
- Trigger “reveal all mines” on loss
- Track win/loss status
- First-clicked cell is guaranteed mine free

2.3 User Interface (UI)

The UI presents the game state visually to the player inside a web browser. It renders the 10x10 grid with column labels (A-J) and row labels (1-10), shows flagged, uncovered, and covered cells, and displays the remaining mine count and current game status (“You Win” or “You Lost”).

Key responsibilities:

- Provide intuitive controls for uncovering and flagging cells
- Display accurate cell contents (number, blank, mine, flag)
- Update dynamically in response to Board Manager and Game Logic state changes

2.4 Input Handler

The Input Handler captures and interprets user actions such as left-click (uncover cell) and right-click (toggle flag). It validates these inputs (ex. preventing uncovering a flagged cell) and forwards them to the Game Logic. It also handles optional keyboard shortcuts if implemented.

Key responsibilities:

- Translate raw user interactions into structured commands
- Prevent invalid moves
- Provide consistent communication between UI and Game Logic

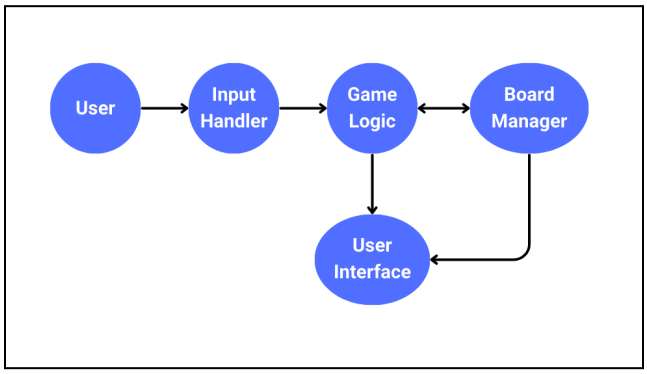

Figure 1: Component Interaction Pathway

Section 3 - Data Flow

3.1 User Input → Input Handler → Game Logic

User Input (ex. clicking a cell) is first captured by the Input Handler. The Input Handler translates the actions by the user into commands that can be processed by the system. If an invalid command is generated, the handler will ignore the input, otherwise valid commands are to be passed to Game Logic. The Game Logic Component will interpret the input given the context of the current state of play (ex. cell uncovering or flag toggling).

3.2 Game Logic → Board Manager

Once the Game Logic interprets a valid user action, board update requests are sent to the Board Manager. The Board Manager is utilized in areas where the state of the grid needs to be adjusted. This includes whether a cell is covered, flagged, or contains a mine. Game Logic works to delegate grid manipulation to the Board Manager. Game Logic maintains the rules of the game, while the Board Manager maintains the underlying data.

3.3 Board Manager → User Interface

The Board Manager supplies the User Interface with the current state of the board. The User Interface requests the necessary details from the Board Manager (ex. cell is hidden, flagged, or uncovered). With incoming data from the Board Manager, the User Interface ensures the user views the board in the most current and relevant state.

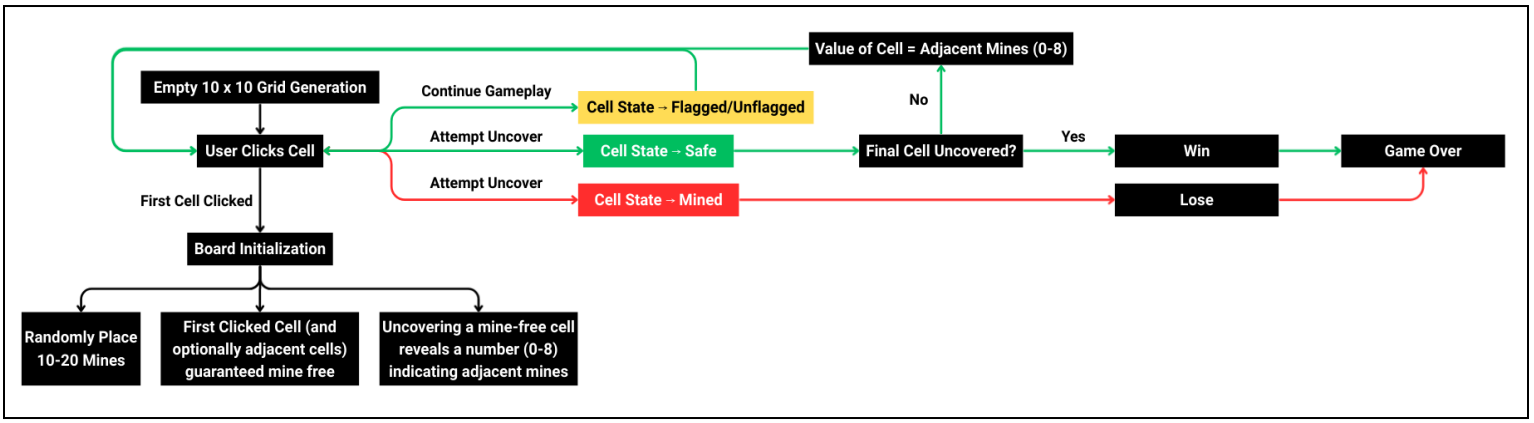

Figure 2: Data Flow Chart

Section 4 - Key Data Structures

4.1 Cell

- Purpose: Unit of game state stored in the board
- Fields (as used across UI/Logic):
  - $isMine$: $boolean$ → whether the square holds a mine
  - $revealed$: $boolean$ → whether the square is uncovered
  - $flagged$: $boolean$ → whether the player marked a flag
  - $adjacent$: $number$ → count of adjacent mines (0-8)

4.2 Board

- Purpose: Authoritative grid of cells and basis for adjacent queries
- Shape: $Cell[][]$ → where this is a 10×10 grid in current configuration
- Provided/used operations:
  - $createEmptyBoard(rows, cols)$ → initialize cells

- $placeMines(board,\ mineCount,\ safeAt?:\ Position)$ → populate mines (first-click safe)
- $computeAdjacency(board)$ → compute adjacent counts
- $cloneBoard(board)$ → immutable-style updates for React state
- $floodFill(board,\ gridSize,\ r,\ c)$ → zero-expansion reveal

4.3 GameState

- Purpose: High-level session state owned by page controller
- Fields:
  - $mines:\ number$ → total number of mines placed at the start of the game
  - $board:\ Cell$ [][] → 2D array holding all cell objects (the board state)
  - $started:\ boolean$ → True once the first reveal is made (used for timing and first-click safe logic
  - $gameOver:\ null \mid 'lost' \mid 'won'$
    - $null$ → game continues
    - $'lost'$ → player reveals a mine
    - $'won'$ → all non-mined cells uncovered
  - $flagsLeft:\ number$ → number of flags still available for the player
  - $seconds:\ number$ → elapsed gameplay time in seconds

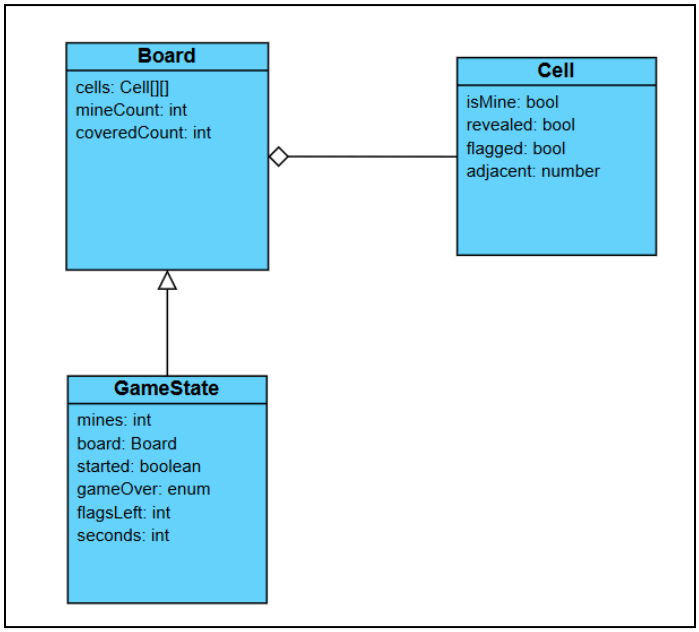

Figure 3: Unified Modeling Language (UML) Diagram for Key Data Structures

System Architecture Document

Section 5 - Assumptions

5.1 Game Setup

- Fixed 10×10 grid size
  - Columns labeled A-J
  - Rows numbered 1-10
- Number of Mines: User specified, 10-20
  - Randomly placed at game start
- First clicked cell (and optionally adjacent cells) guaranteed mine-free
- Initial State: All cells start covered with no flags

5.2 Gameplay

- Players uncover a cell by selecting it (clicking)
- Uncovering a mine ends the game in a loss
- Uncovering a mine-free cell reveals a number (0-8) indicating adjacent mines
- Cells with zero adjacent mines trigger recursive uncovering of adjacent cells
- Players can toggle flags on covered cells to mark suspected mines

5.3 Mine Flagging

- Players place/remove flags on covered cells to indicate potential mines
- Flagged cells cannot be uncovered until unflagged
- Display remaining flag count ($total\ mines\ -\ placed\ flags$)

5.4 Player Interface

- Display a 10×10 grid showing cell states:
  - covered
  - flagged
  - uncovered (number or empty for zero adjacent mines)
- Show remaining mine count ($total\ mines\ -\ total\ flags$)
- Provide a status indicator for gameplay

5.5 Game Conclusion

- Loss: Triggered by uncovering a mine, revealing all mines
- Win: Achieved by uncovering all non-mine cells without detonating any mines

System Architecture Document

Section 6 - Person-Hours Estimation

6.1 Estimation Methodology

For the person-hours estimate, a structured approach was used to evaluate a probable length of time for determining task completion. This structure relied on experience in previous software engineering projects, ranging from real-world application to school applications. This was especially useful for areas related to team member contributions and dynamics. For each task, time estimates were derived using references from previous projects allowing for realistic estimations. For areas where historical reference was not applicable, a consensus-based method was used to provide informed estimates. For project uncertainties and external implications, additional time was added to ensure the overall time allocation remained practical.

6.2 Person-Hours Estimate Table

The Person-Hours Estimate Table is located in the GitHub Repository.

Section 7 - Actual Person-Hours Accounting

7.1 Actual Person-Hours Table

The Actual Person-Hours Accounting Table is located in the GitHub Repository.