EECS 581 — Project 1: Minesweeper System Documentation

Team 26

September 19, 2025

Synopsis

This project implements classic Minesweeper in Python with Pygame on a fixed 10x10 grid with a user selected 10 to 20 mines, using a minimal yet modular architecture of four components: Input Handler (captures and validates user actions), Game Logic (safe first click, uncover and flag rules, zero cell flood fill, win or loss detection), Board Manager (owns the grid, places unique mines, computes neighbor counts, tracks cell states), and User Interface (renders board, counters, and status). Data flows from user actions to the Input Handler, then to Game Logic which queries and updates the Board Manager, after which the UI renders the updated state. Each cell records mine presence, covered or uncovered state, flagged state, and adjacent mine count, while a central game state tracks total mines, flags remaining, covered safe cells, and overall status. The lifecycle proceeds from START to PLAYING, then to WIN or LOSS, with reset returning the game to START. This separation of concerns keeps our codebase small and enables future extensions by further teams, such as difficulty presets or alternate front-end designs.

1 System Architecture Overview

1.1 Purpose

The purpose of this architecture is to keep Minesweeper small, testable, and easy to extend by cleanly separating the game module (state and rules) from the frontend (rendering and input). This separation enforces a simple, one-way flow of data—user input to input handling to game logic and board state, then to the user interface—so correctness of rules such as safe

first click, flag validation, breadth-first reveal of zero cells, and win or loss detection can be verified independently of rendering. Our design uses explicit states (Start, Playing, Win, Loss) and a minimal set of data structures (Cell, Grid, GameState) to make the code understandable and maintainable. Overall, the goal is clarity for our reviewers and maintainers, reliable gameplay under stress, and a smooth handoff for the next part of our project. Figure 1 shows the methodical connections between our components, and will help to showcase how we accomplish our purpose.

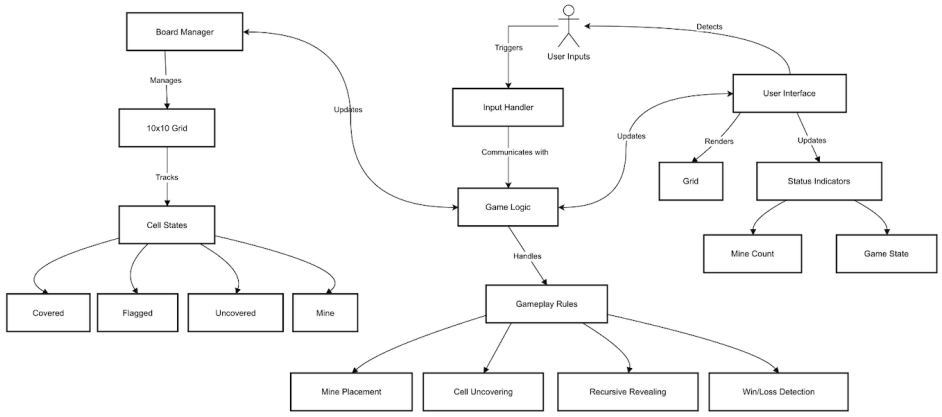

Figure 1: Full component interaction pathways

1.2 Components

This system architecture pertains to four main components and their interactions:

- **User Interface (UI):** Renders the grid, remaining mines/flags, and overall status based on the current game state. Reflects updates from the game and exposes only the controls appropriate to the active state.
- **Input Handler:** Captures mouse position, clicks, and keyboard input, classifying them into high-level actions (uncover, toggle flag, reset, set mine count). Gating is state-aware: board actions are permitted only during *PLAYING*, while mine count entry is limited to *START*; invalid actions are ignored before reaching the game.
- **Game Logic:** Implements the state machine ($START \rightarrow PLAYING \rightarrow END::WIN/LOSS \rightarrow START$) and enforces rules. Ensures first

click safety, places unique mines, computes neighbor counts, performs BFS flood-reveal for zero cells, updates counters, validates flags, and detects win/loss.

- **Board Manager:** Owns the 10x10 grid and per-cell data (is_mine, covered, flagged, neighbor_count). Provides helpers for neighbor enumeration, unique mine placement, board reset, and a simple query/update API for logic and rendering.

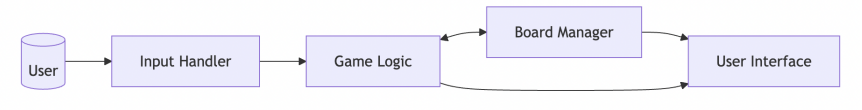

Figure 2: Component Interaction Diagram

1.3 Data Flow

- **User Action** → Mouse/keyboard input is captured by the frontend; inputs allowed depend on the current state (`START`, `PLAYING`, `END::WIN/LOSS`).
- **Input Handler** → Normalizes and validates events (e.g., uncover, toggle flag, reset, set mine count); invalid or state-blocked actions are ignored; valid actions are emitted as commands.
- **Game Logic** ↔ Applies rules to the current state: first-click safety, unique mine placement, neighbor counts, BFS zero-cell reveal, flag validation, counters, and win/loss detection. Our mine placement uses RNG (random number generation).
- **Board Manager** ↔ Executes queries/updates from Game Logic: sets/unsets `is_mine`, (re)computes `mine_count`, flips `covered`/`flagged`, and returns a snapshot of `Grid`/`Cell`/`GameState` to our user.
- **UI Update** → The frontend reads the latest snapshot and re-renders the board and indicators (remaining mines/flags, status), enabling only the controls appropriate to the active state; the loop then awaits the next user action. This process is (simply) visualized in Figure 5.

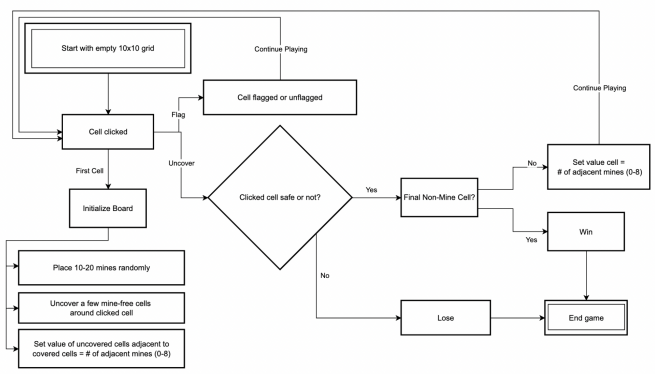

Figure 3: Game Flow Diagram

1.4 Key Data Structures

- **Cell:** Holds per-tile state: `is_mine`, `covered`, `flagged`, and `neighbor_count` (0–8). Game Logic flips these fields on uncover/flag actions and maintains the invariant that a cell cannot be both flagged and uncovered.
- **Grid:** A fixed 10x10 array of **Cell** that provides neighbor enumeration (8 directions), unique mine placement on the first click, neighbor count computation, and full reset for a new game. It also maps user coordinates to indices for consistent input handling. There is room for change here during Project 2.
- **GameState:** Tracks global `status` (`START`, `PLAYING`, `END::WIN`, `END::LOSS`), `total_mines` (total mines), `flags_remaining` (current count), and `covered_safe_cells` (safe cells still covered), with potential for optional flags or mine seeding upon development during Project 2. As the state is updated, the board should reflect the updates.

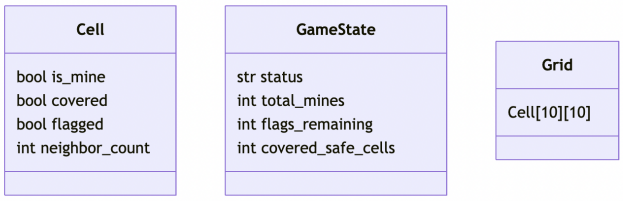

Figure 4: Data structures tabular representation

1.5 Assumptions & Constraints

Assumptions

- Fixed board size is 10x10; columns labeled A-J and rows 1-10.
- The user selects the mine count at start; valid range is 10–20 and is validated before play begins.
- First click is always safe: mines are placed (or adjusted) on the first uncover so the first clicked cell is not a mine.
- Adjacency uses 8 neighbors for computing mine count.
- Cell states are discrete and mutually consistent: `covered` vs. not; `flagged` implies not uncovered.
- Game states are `START` $\rightarrow$ `PLAYING` $\rightarrow$ `END::WIN/LOSS` (with reset returning to `START`).
- Primary inputs are mouse clicks for board actions and a simple keyboard field for mine count entry.

Constraints

- UI must be intuitive without a manual and display at least: grid, remaining mines (or flags), and current status (`Playing`, `Win`, `Loss`).
- Performance/stability: the game must withstand rapid input (stress testing) without crashes or memory leaks; rendering should remain responsive.
- Rules integrity: flagged cells cannot be uncovered; `flags_remaining` stays within $[0, \texttt{total\_mines}]$; `remaining` decrements only when a safe cell is uncovered; mines occupy unique cells.
- Scope limits (v1): only the specified 10x10 board; no alternative board sizes, timers, scoring, or difficulty presets beyond the user-selected mine count.
- Process: demo must run from the `master` branch as of the code-freeze commit; system documentation and diagrams reside in the repository.
- Testing: unit tests focus on first-click safety, neighbor counts, BFS zero-area reveal, flag gating, and win/loss detection.

1.6 Extension Points

Our group has included points of code that have great potential for upgrades, or new features. We've listed some:

- Difficulty preset: Consider using our method of determining total mines to pre-calculate difficulties that increase the amount of mines in an area, above our current mine limit.
- Seed play: Instead of using RNG to generate a board each time, create a seed method that creates a board (some numerical combination), and anytime the seed is referenced, reproduces said board.
- UI change: Consider altering the UI to better suit your group, display the play status differently, or display an entirely new statistic.

2 Person-Hours Estimate (Methodology)

2.1 Estimation Technique

We used a quantitative, team-based story point process. After decomposing the project into 27 tasks, each member estimated points (planning-poker style) and we took the consensus/median for the final value per task. We then prioritized the backlog with emphasis on higher-value and higher-risk items. In parallel, we computed a Use Case Points proxy: $UUCP = UUCW + UAW = 70$, and applied a focus factor of 0.55 (time-boxed schedule, simple domain), yielding a working capacity of ≈ 38.5 team-points for the timebox. This capacity informed load balancing across six members and daily targets; we used it as guidance rather than a hard hours conversion to keep points relative.

2.2 Assumptions

We assume one synchronous Scrum meeting per week and 24/7 asynchronous coordination via group chat ("on call"). Team members complete delegated work, adhere to the rubric, and respect the code-freeze and demo requirements (master branch at the freeze commit).

2.3 Risks & Buffers

Primary risks include correctness edge cases (first-click safety, BFS zero-region reveal, neighbor counts), input/UX gating by state, and stability

under rapid input (performance or leaks). We reserve a stabilization buffer (10–15% of the timebox), rely on deterministic tests, and schedule code reviews. If this is not enough to mitigate all the risk, we can also meet with our Scrum Master to discuss it further.

2.4 Estimated Hours by Task

Estimated hours per task are maintained in the repository alongside the backlog. The distribution follows the technique above (consensus story points, priority by value/risk) and reflects the 38.5 team-point capacity for the timebox.

3 Actual Person-Hours (Day-by-Day)

3.1 Daily Log

Actual hours are recorded in the repository using two spreadsheets stored side by side: the Estimation log and the Actual Hours log. Each team member updates the Actual Hours log daily with brief entries (date, member, task, start/end, duration in integer increments). Project work only is counted (coding, testing, documentation, meetings); course lecture time is excluded. The sheet totals hours per week and project to date for each member. Reviewers can verify entries against visible project activity (commits near the code-freeze, pull requests, and demo readiness). This log is used for accountability and load balancing.

4 System Diagrams

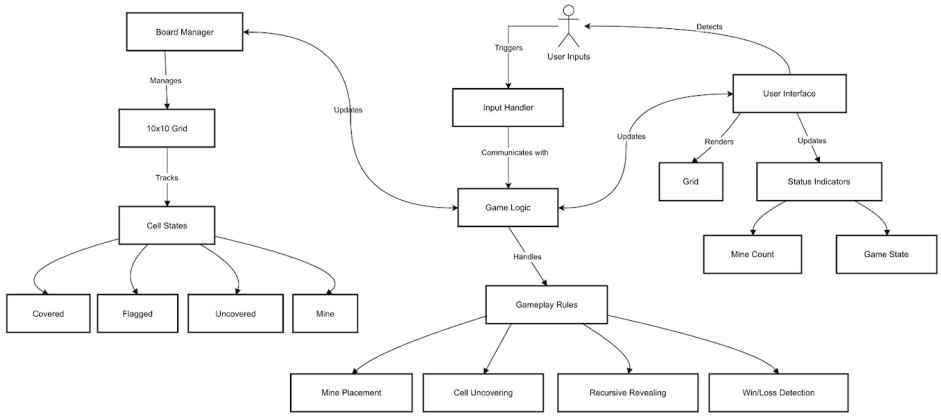

Figure 1. Full component interaction pathways (reproduced)

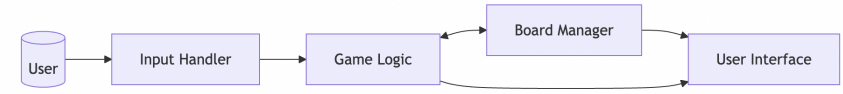

Figure 2. Component Interaction Diagram (reproduced)

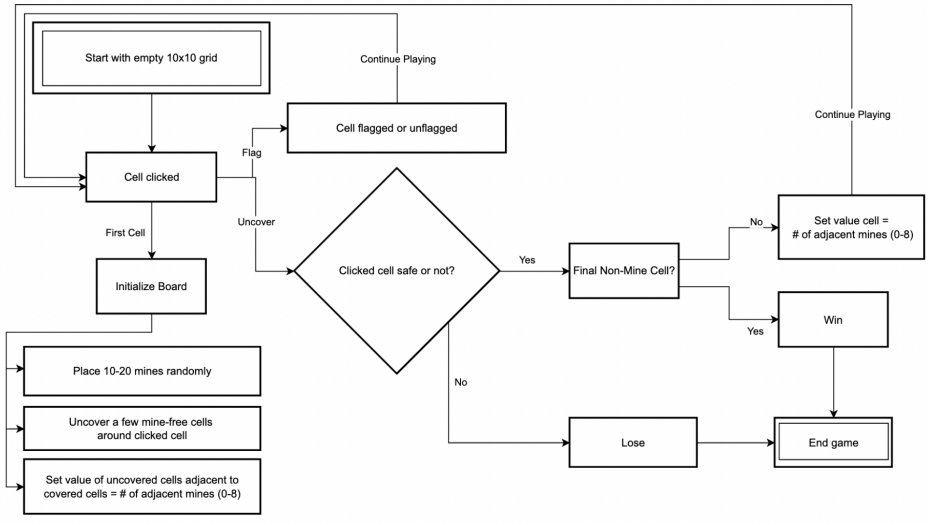

Figure 3. Game Flow Diagram (reproduced)

Cell
bool is_mine bool covered bool flagged int neighbor_count

GameState
str status int total_mines int flags_remaining int covered_safe_cells

Grid
Cell[10][10]

Figure 4. Data structures tabular representation (reproduced)

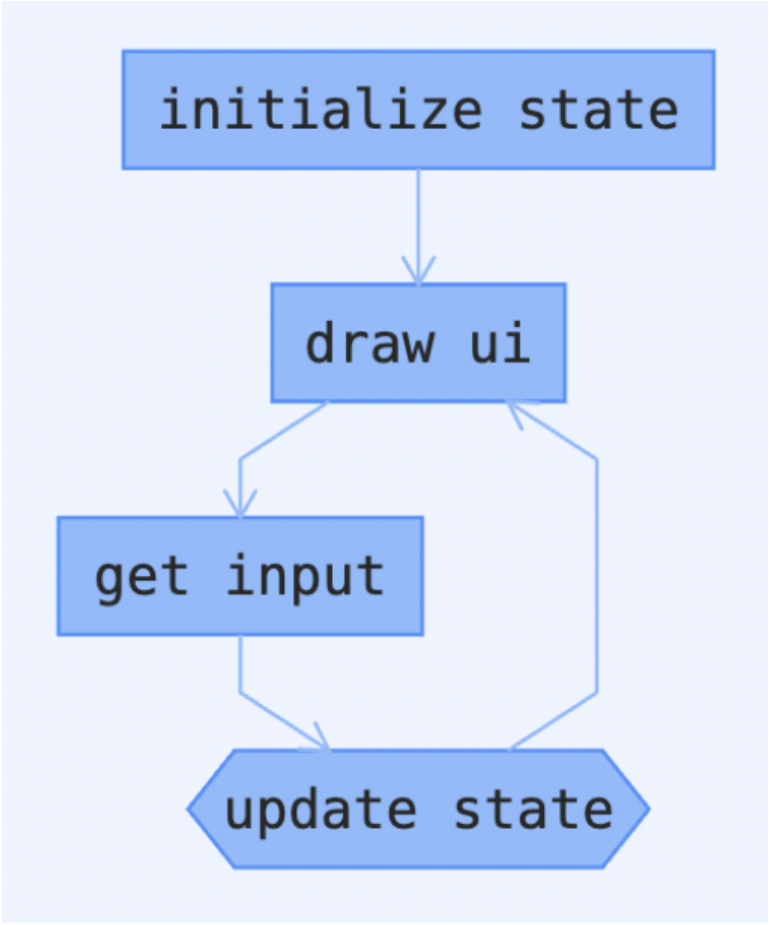

Figure 5: Simple UI diagram to aid in core understanding