Project 2: Maintenance System Documentation

Person-Hours Estimate (10 Points):

Methodology: Use Case Points:

Ensure Project 1 Features are Fully Operational: (2 steps)

Complexity: Simple

Weight: 5 points

Artificial Intelligence Solver: (3 steps)

Complexity: Simple

Weight: 5 points

Custom Additions: (3 steps)

Complexity: Simple

Weight: 5 points

Total Unadjusted Use Case Weight: 15 points

Actors:

User: A person playing the game

Complexity: Complex

Weight: 3 points

Total Unadjusted Actor Weight: 3 points

Unadjusted Use Case Points:

15 +3 = 18 points

Final Effort:

18 Use Case Points x 1 person hour per UCP = 18 person-hours

Actual Person-Hours:

Cole:

- (9/24/2025): Spent 1 hour reviewing inherited project and code
- (9/30/2025): Spent 2.5 hours working on AI Solver module and implementation into main
- (10/3/2025): Spent 45 minutes completing medium ai difficulty implementation

Riley:

- (9/30/2025): Spent 1 hour reviewing inherited project and code
- (9/30/2025): Spent 1.5 hours working on difficulty levels and implementation into main
- (10/5/2025): Spent 45 minutes reviewing code and finishing system documentation

Manu:

- (9/24/2025): Spent 30 min setting up the original code and testing efficacy of it.
- (9/1/2025): Spent 1 hour implementing hint functionality.
- (10/5/2025): Spent 1 hour on prologue comments and system documentation

Evans:

- (09/26/2025): Spent 1.5 hours reviewing the inherited project and researching sound implementation in Pygame.
- (09/30/2025): Spent 1.5 hours implementing sound effects into the game.
- (10/5/2025) : Spent 1 hour integrating the sound effects into the main file and finishing the system document.

Jackson:

- (9/24/2025): Spent 1 hour finding bugs and brainstorming possible fixes.
- (9/30/2025): Spend 0.5 hour reviewing codebase and testing bugs.
- (10/5/2025): Spent 1.5 hours fixing remaining bugs and pushing them to github.

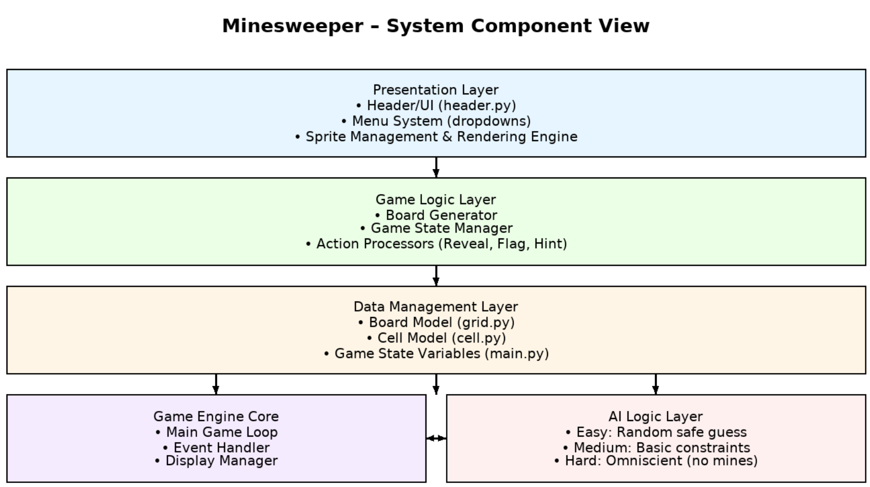
System Architecture Overview:

The Minesweeper game project is implemented using Python and Pygame. The system uses a modular architecture with separation between game logic, state management, and presentation layers. The application operates on a game loop pattern, continuously processing user input events and updating the display accordingly.

Key Technologies:

- Python 3.x
- Pygame (rendering, events, sprites)
- Pygame-widgets (dropdowns)

System Components:



Data Management Layer

- Board Model (grid.py): Manages the core game data structure (Grid). Implements methods for coordinate translation, cell creation, mine placement, adjacency computation, and flood-reveal.
- Cell Model (cell.py): Represents each tile as a pygame.sprite.Sprite with attributes for bombs, flags, reveal state, and nearby mine count. Handles all per-cell rendering logic and animations.
- Game State Variables (main.py): Tracks menu/game/win/loss states, bomb counts, flags, hints, and timers.

Presentation Layer

- Sprite Management: Loads and manages visual assets. Each cell dynamically updates its texture based on game state.
- Rendering Engine: Draws game board, labels, cells, hint button, and visual setup.
- Header/UI (header.py): Provides a visually pleasing top bar displaying time elapsed, remaining mines, and decorative assets.
- Menu System: Main menu includes dropdowns for AI difficulty, mode (auto or interactive), bomb count, and difficulty preset (easy, medium, hard). Clicking "Start" initializes the game with those selections.

Game Logic Layer

- Board Generator: Builds a grid of cells, randomly places bombs, and calculates adjacent mine counts.

- Game State Manager: Continuously checks for win and loss conditions and freezes the timer on completion.
- Action Processors:
  - Reveal: Reveals a selected cell; if it has zero adjacent bombs, recursively reveals its neighbors.
  - Flag: Toggles flagged status for cells; updates remaining counter.
  - Hint System: Reveals one safe cell per use(up to 3 hints per game).

Game Engine Core

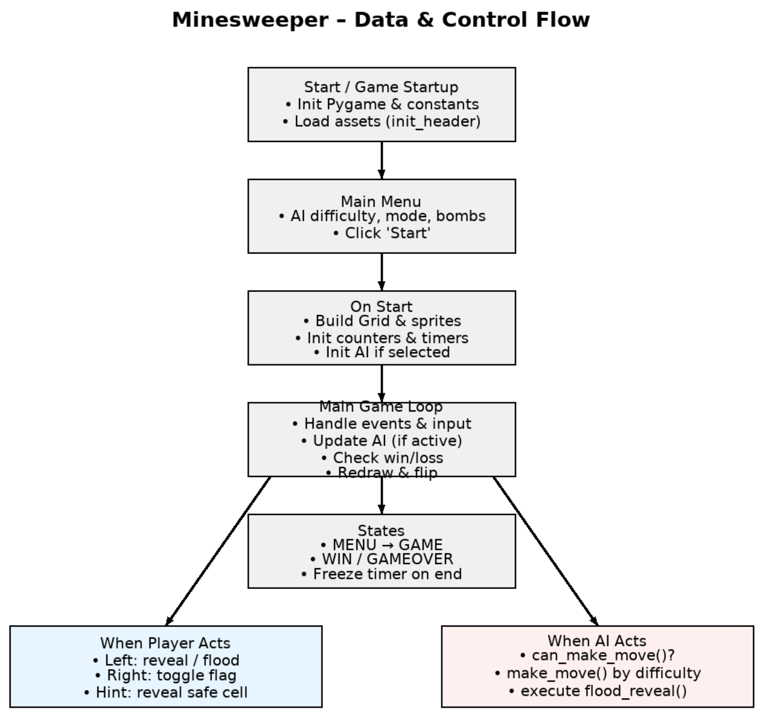
- Main Game Loop: Coordinates event handling, updates AI turns, tracks player input, manages drawing cycles, and triggers win/loss states.
- Event Handler: Processes user input (mouse clicks, keyboard input) and menu interactions.
- Display Manager: Handles screen rendering and visual updates; recomputes display scaling per difficulty preset.

AI Logic Layer

- Implements a three-tier rule-based solver:
  - Easy Mode: Selects random unrevealed and unflagged cells.
  - Medium Mode: Flags hidden neighbors if their count equals a revealed number and uncovers safe cells when flagged neighbors equal that number.
  - Hard Mode: Extends Medium logic but avoids mines completely by accessing where mines are placed.

- Turn Management: In auto mode, the AI moves continuously. In interactive mode, turns alternate between the player and AI.

Data Flow



Game Startup

1. Import libraries, initialize Pygame, and define constraints.
2. Load assets via init_header().
3. Display the main menu with dropdowns for AI settings and difficulty.
4. On start:
    - startGame() constructs a new Grid and window based on the selected difficulty.
    - Creates and positions cell sprites.
    - Initializes counters, timers, and AI if selected.

5. Enter the main loop.

Every Frame in Main Loop

1. If MENU: draw start menu and update widgets
2. If GAME: process inputs (reveal, flag, hint), update AI turn if active, check win/lose, and redraw the board and the header.
3. If WIN or GAMEOVER: display end screen and freeze timer.
4. Refresh display with pygame.display.flip()

When Player Acts

- Left Click: Mouse position → Convert to grid coordinates → Check if valid → Call grid.flood.revel() → Update revealed array
- Right Click: Mouse position → Convert to grid coordinates → Call grid.flag()→ Toggle flag and adjust buoys_left.
- Hint Button: Call use_hint() to reveal one safe cell and decrement hints.

When AI Acts

- Main loop checks can_make_move() after a short delay.
- Calls make_move() according to selected difficulty.
- Executes grid.flood_revel() on chosen coordinates.
- In interactive mode, toggles turn control back to the player.

Key Data Structures:

Primary Data Arrays:

Board Array (board: List[List[int]])

- Type: 2D integer matrix (SIZE x SIZE)
- Purpose: Stores the core game state as numbers per tile.
- Values:
  - -1: Mine location
  - 0 - 8: Number of adjacent mines.
- Lifecycle: Generated once per game after bomb placement and neighbor counts. Remains constant.

Revealed Array (revealed: List[List[int]])

- Type: 2D boolean matrix (SIZE x SIZE)
- Purpose: Tracks which cells are currently visible to the player.
- Values:
  - True: Cell is revealed
  - False: Cell is hidden.

Flagged Array (flagged: List[List[int]])

- Type: 2D boolean matrix (SIZE x SIZE)
- Purpose: Tracks which cells have been flagged by the player.
- Values:
  - True: Cell is flagged.
  - False: Cell is not flagged.

Configuration Constants:

- HEADER_HEIGHT: int = 150 # Header bar height
- CELL_PIXELS: int = 50 # Pixel size of each cell
- gameHeight: int = 10 # Height of game window
- gameWidth: int = 10 # Width of game window
- gameSize: int = CELL_PIXELS * gameWidth #Size of the game
- padding: int = 50 # Outer padding around the board
- displaySize: int = gameSize + padding * 2 # Size of the display
- WINDOW_HEIGHT = HEADER_HEIGHT + displaySize # Height of game window
- stop_time = 0 # To freeze timer on win/lose
- bombs_count: int | None # From ‘Bombs’ Dropdown, overrides density if set

Game State Variables

- # High level state flags
    - MENU: bool
    - GAME: bool
    - WIN: bool
    - GAMEOVER: bool
- # Timing & turns
    - start_time: float
    - stop_time: float
    - ai_mode: str | None # ‘None’ | ‘Easy’ | ‘Medium’ | ‘Hard’
    - game_mode: str # ‘Auto’ | ‘Interactive’
    - player_turn: bool

- # Counters and options
    - hints_remaining: int = 3
    - buoys_left: int   # remaining flags shown in header
    - bombs_count: int | None   # explicit count (overrides density)

Sprite Dictionary

- num_sprite = {

    1: ‘textures/Tile_1.png’,   # Cells with 1 adjacent mine

    2: ‘textures/Tile_2.png’,   # Cells with 2 adjacent mines

    ….

    8: ‘textures/Tile_8.png’,   # Cells with 8 adjacent mines

  }
- state_sprite = {

    ‘unrevealed’: ‘textures/Unrevealed_Tile.png’,

    ‘revealed’: ‘textures/Revealed_Tile.png’,

    ‘flagged’: ‘textures/Flagged_tile.png’,

    ‘mine’: ‘textures/Mine_Tile.png’

  }
- Header textures: Flag_Header.png, Mine_Header.png, planks.png, wheel.png, compass.png, buoy.png
- End screens: Win_Screen.png, Lose_Screen.png