Minesweeper Architecture Overview (Project 2)

This Minesweeper implementation follows a modular, layered architecture with clear separation of concerns. The project uses Python and Tkinter to create a GUI game with abstractions between game logic, user interface, data management, and AI components.

ARCHITECTURE PRINCIPLES

1. Separation of Concerns

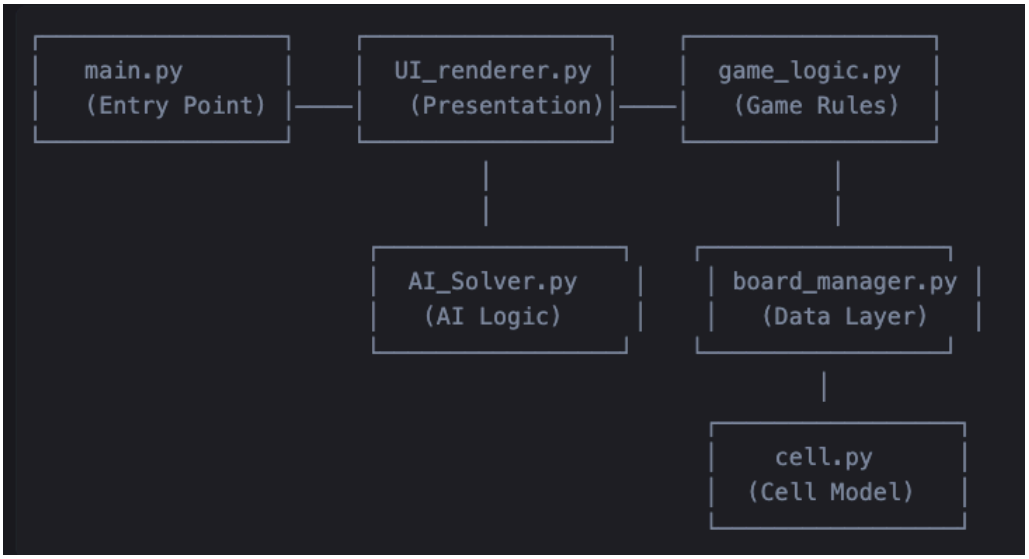
- Game Logic: Pure game rules and state management
- UI Layer: Visual presentation and user interaction
- Board Manager: Data structure and mine placement algorithms
- Cell Model: Individual cell state representation
- AI Solver: Computer opponent logic

2. Event-Driven Design

The application responds to user events (clicks, mode toggles) through Tkinter's event system, ensuring responsive user interaction without blocking the UI thread.

3. Dependency Injection

Components receive their dependencies through constructor injection, making the system testable and loosely coupled.



CORE COMPONENTS

1. main.py - Application Entry Point
Responsibility: Initialize the GUI and start the application
Key Functions:
- Creates the GameGUI instance
- Starts the Tkinter main loop
Dependencies: UI_renderer.py

2. cell.py - Cell Model
Responsibility: Represent individual board cells with their state
Key Classes :
- Cell: Represents individual board cells
Key Attributes :
- has_mine: Boolean indicating mine presence
- has_flag: Boolean indicating flag status
- is_revealed: Boolean indicating reveal status
- neighbor_count: Number of adjacent mines (0-8)
Key Methods :
- flag(): Mark cell as flagged
- unflag(): Remove flag from cell
- add_mine(): Place mine in cell
- remove_mine(): Remove mine from cell

3. board_manager.py - Data Layer
Responsibility: Manage game board data structure and mine placement
Key Classes :
- BoardManager: Manages the 2D board array and mine placement algorithms
Key Methods :
- __init__(grid_size, mine_count): Initialize empty board
- place_mines(safe_row, safe_col): Place mines with a safe first-click zone
- get_cell(row, col): Access individual cells
- neighbors(row, col): Get adjacent cell coordinates
- count_adjacent_mines(row, col): Calculate neighbor mine count
- untouched_cells(): Get unrevealed/unflagged cells
- reset(mine_count): Clear board for new game

4. game_logic.py - Business Logic Layer
Responsibility: Implement game rules, state transitions, and victory conditions
Key Classes :
- GameLogic: Core game state management
Key Methods :
- reveal_cell(row, col): Handle cell reveals with flood-fill

- toggle_flag(row, col): Handle flag placement/removal
- reset_game(mine_count): Initialize new game
- _flood_reveal(row, col, out_list): Recursive reveal algorithm
- _all_mines_flagged(): Check perfect flag placement
- easy(reveal, setFlag): AI easy difficulty
- medium(reveal, setFlag): AI medium difficulty

5. UI_renderer.py - Presentation Layer

Responsibility: Create and manage the Tkinter GUI, handle visual feedback

Key Classes :

- GameGUI: Main GUI controller

Key Components :

- Game Board (10x10 grid of buttons)
- Status Display (timer, turn counter)
- Control Panel (flag toggle, mine count input)
- Dialog System (game over, victory messages)

Key Methods :

- renderBoard(): Create button grid
- renderCell(row, col, flag): Update cell appearance
- reveal(row, col): Handle cell clicks
- addFlag(row, col): Handle flag toggling
- toggleFlag(): Switch between flag/reveal modes
- startGame(): Initialize new game
- start_timer(): Begin game timer
- updateStatus(status): Update status display

6. AI_Solver.py - AI Logic Layer

Responsibility: Implement a computer opponent with different difficulty levels

Key Classes :

- AISolver: Computer opponent controller

Key Methods :

- play_turn(): Execute AI move
- easy(): Random cell selection
- medium(): Basic rule-based strategy
- hard(): Advanced strategy (placeholder)

DATA FLOW

Initialization Flow:

main.py → GameGUI → BoardManager → GameLogic → Cell objects

User Interaction Flow:

User Click → UI_renderer.reveal() → GameLogic.reveal_cell() → BoardManager.get_cell() → UI_renderer.renderCell()

Flag Toggle Flow:
User Click (Flag Mode) → UI_renderer.addFlag() → GameLogic.toggle_flag() → UI_renderer.renderCell()

AI Turn Flow:
GameLogic → AISolver.play_turn() → GameLogic.easy/medium() → UI_renderer.reveal()

STATE MANAGEMENT

Game States:
- Initialization: Setting up board and UI
- Playing: Active gameplay
- Game Over: Mine hit or victory
- Victory: All safe cells revealed

Cell States:
- Covered: Default state, not revealed
- Revealed: Clicked and showing neighbor count
- Flagged: Marked with flag by player
- Mine: Contains mine (revealed on game over)

Game Logic State:
- is_first_click: Boolean for safe first-click mine placement
- is_game_over: Boolean for game termination
- revealed_safe_cells: Counter for victory condition
- flags_placed: Counter for flag limit enforcement
- total_safe_cells: Calculated target for victory

ERROR HANDLING

Input Validation:
- Mine count validation (10-20 range)
- Grid bounds checking
- Flag limit enforcement

Exception Handling:
- ValueError for invalid mine counts
- IndexError for out-of-bounds cell access
- Game state validation in logic layer

TESTING

Manual Testing:
- Flag toggle functionality
- Cell reveal behavior
- Mine detection
- Victory conditions
- AI opponent behavior

Integration Points:
- UI ↔ Game Logic communication
- Board Manager ↔ Cell state synchronization
- AI Solver ↔ Game Logic coordination

DEPLOYMENT

Dependencies:
- Python 3.x
- Tkinter
- Standard library modules (time, random, typing)

Distribution:
- Standalone Python application
- Cross-platform compatibility
- No external dependencies required