Report : Minesweeper Game

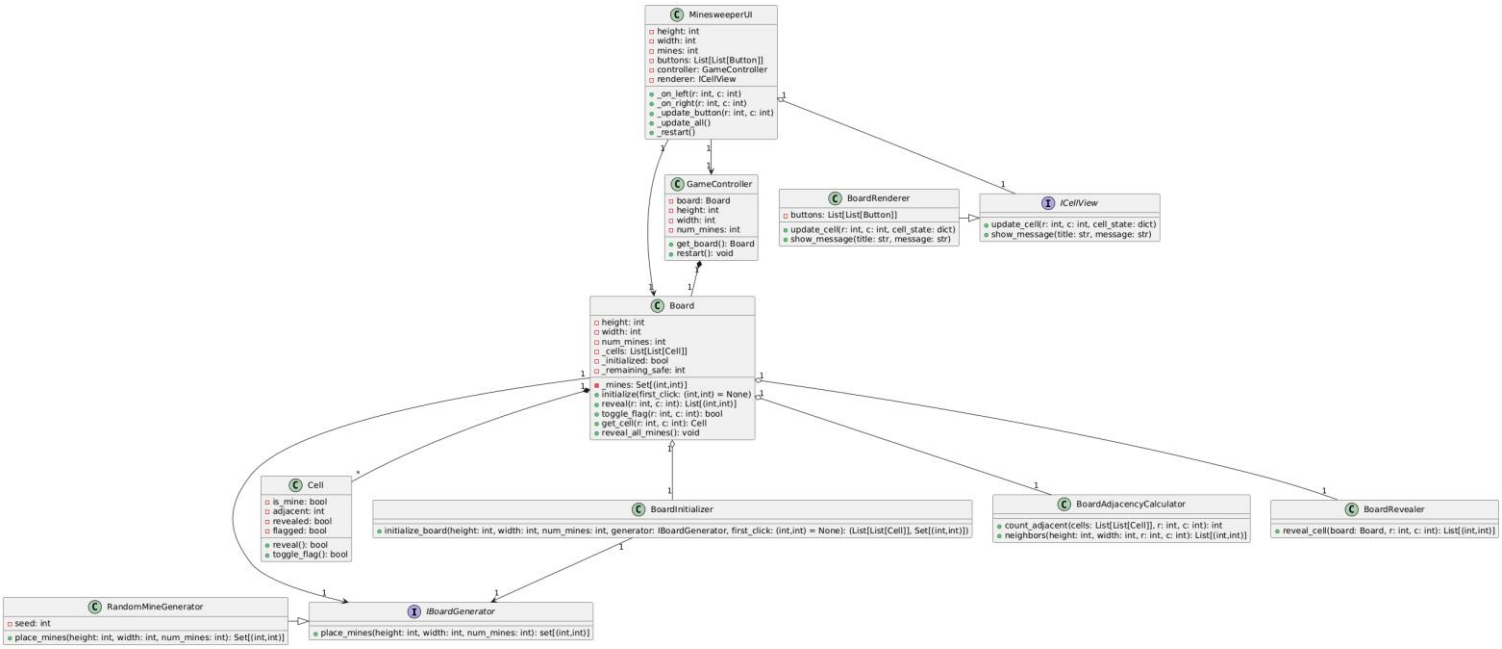
1. Introduction

This project is an object-oriented implementation of the Minesweeper game in Python. The objective of the game is to reveal all cells that do not contain mines. When a player clicks a cell:

- If it contains a mine, the game ends in defeat.
- Otherwise, the number of adjacent mines is shown.
- The player can also flag cells suspected to contain mines.

The program follows a modular and extensible design based on object-oriented programming (OOP) principles and adheres to SOLID design principles to ensure maintainability, scalability, and clarity.

2. UML diagram



3. Class Descriptions and Responsibilities

Below is a description of the classes and their function in the creation of the game.

GameController:

The GameController class is responsible for orchestrating the overall game flow. It creates and manages the Board instance and acts as a bridge between the game logic and the user interface. It encapsulates the game setup and control logic so that the UI doesn't need to directly manipulate the board.

Method:

- restart() : creates a new Board with the same configuration (height, width, and number of mines) to start a new game.

Board:

The Board class represents the game's core logic: it manages all cells, mine placement, and the rules of Minesweeper. It's the central piece of game logic but delegates specific responsibilities to helper classes to remain clean and modular.

It collaborates with:

- BoardInitializer: Sets up the grid and places mines safely (ensuring the first click is never a mine).
- BoardAdjacencyCalculator: counts adjacent mines for each cell.
- BoardRevealer: handles the recursive reveal of empty cells.

Methods:

- initialize(first_click=None): Initializes the grid and mine placement.
- reveal(r, c): reveals a cell and possibly triggers a flood fill if the cell has no adjacent mines.
- toggle_flag(r, c): flags or unflags a cell.
- get_cell(r, c): returns the Cell object at given coordinates

Cell:

The Cell class models a single square on the Minesweeper grid. It stores its own state (mine, revealed, flagged, number of adjacent mines). Each cell knows its own state but doesn't know about the board as a whole.

Methods:

- reveal(): marks the cell as revealed (unless flagged).
- toggle_flag(): switches the flagged state on or off.

BoardRenderer:

The BoardRenderer is responsible for updating the graphical representation of the game board. It acts as the visual adapter between the Board logic and the Tkinter user interface. It updates button texts and states in the UI when the board changes.

Methods:

- update_cell(r, c, cell_state) → Updates a single cell's display (revealed number, mine, or flag).
- show_message(title, message) → Displays game outcome messages (e.g., Victory or Defeat

MinesweeperUI:

The MinesweeperUI class manages the graphical user interface using Tkinter, Python's built-in GUI toolkit. It provides the interactive window where players click cells, flag mines, and start new games.

Tkinter is included in Python by default. It allows quick creation of buttons, labels, and menus without additional dependencies.

Methods:

- _on_left(r, c): handles left-click actions to reveal cells.
- _on_right(r, c): handles right-click actions to toggle flags.
- _update_button(r, c): refreshes a specific cell's appearance.
- _restart(): calls the controller to reset the game.

The interfaces:

- **IBoardGenerator:**
  Defines the interface responsible for generating mine positions on the board, allowing different mine placement strategies (random, seeded, deterministic, etc.) to be implemented interchangeably.
- **IBoardLogic:**
  Specifies the core logical operations that any game board must implement, such as revealing cells, toggling flags, and accessing cell information.

- **ICellRenderer:**
  Provides the interface for updating the user interface (UI) in response to changes in the game board, such as revealing or flagging a cell, or showing win/lose messages

4. Application of SOLID Principles For Each Class

GameController class

The GameController class respects the Single Responsibility Principle because it is only responsible for managing the game flow. It initializes and restarts the game without handling any game logic or UI rendering. It also follows the Open/Closed Principle since new game modes or rules can be added by extending the controller without modifying the existing code. The Liskov Substitution Principle is respected because any other controller could replace it as long as it performs the same role. The class also follows the Interface Segregation Principle as it only depends on the board interface and does not need to implement or use unnecessary methods. Finally, it follows the Dependency Inversion Principle because it depends on abstractions (the board logic interface) rather than concrete implementations.

Board class

The Board class follows the Single Responsibility Principle because it focuses on the core Minesweeper game logic. It manages the cells and rules of the game but delegates specific tasks such as initialization, adjacency calculation and recursive reveal to separate helper classes. It respects the Open/Closed Principle as its behavior can be extended, for example by adding new board rules, without modifying the existing code. The design also follows the Liskov Substitution Principle since any alternative implementation of the board, as long as it follows the board logic interface, can replace it without affecting the rest of the system. The Board respects the Interface Segregation Principle by exposing only the necessary methods for interacting with the game (such as reveal, toggle flag, and get cell). Finally, it follows the Dependency Inversion Principle because it depends on abstractions such as the IBoardGenerator interface rather than directly depending on concrete classes.

Cell class

The Cell class strongly follows the Single Responsibility Principle because it is only responsible for storing and managing the state of one cell. It does not handle any external game rules or board logic. The Open/Closed Principle is respected because the class can be extended to include new features (for example a “question mark” state) without modifying its original code. It also respects the Liskov Substitution Principle since a different cell implementation can replace it if it respects the same behavior. The Interface Segregation Principle is naturally respected since the class only exposes two simple methods directly

related to its state. The Dependency Inversion Principle does not really apply to this class because it does not depend on other components.

BoardRenderer class

The BoardRenderer class respects the Single Responsibility Principle since it only manages the graphical representation of the board and does not contain game logic. It follows the Open/Closed Principle because new renderers can be added (such as a console renderer or a web renderer) without changing the current class. It respects the Liskov Substitution Principle because any class implementing the renderer interface can replace it. It also respects the Interface Segregation Principle by depending only on the methods required to update the UI. Finally, the Dependency Inversion Principle is respected because the UI depends on the ICellRenderer abstraction rather than on the concrete renderer implementation.

MinesweeperUI class

The MinesweeperUI class respects the Single Responsibility Principle because it is only responsible for managing the graphical user interface and user interactions through Tkinter. It does not perform any game logic itself. It follows the Open/Closed Principle because the UI can be extended to support new features (such as themes or additional menus) without modifying existing code. The Liskov Substitution Principle is also respected as a different type of UI (such as a web or mobile UI) could replace it while keeping the controller and game logic unchanged. The Interface Segregation Principle is respected because the UI only depends on the methods required for rendering and interacting with the game. It also respects the Dependency Inversion Principle because it communicates with the controller rather than directly manipulating the board.

Interfaces

IBoardGenerator interface

The IBoardGenerator interface follows the Single Responsibility Principle by defining only the functionality needed for generating mine positions. It also respects the Open/Closed Principle since new mine generation strategies can be created without modifying the interface. The Liskov Substitution Principle is respected as any implementation can substitute another. The Interface Segregation Principle is followed because the interface remains small and focused. It also supports the Dependency Inversion Principle by allowing the system to rely on the abstraction and not a specific generator.

IBoardLogic interface

The IBoardLogic interface also follows the Single Responsibility Principle because it defines only the essential operations that a Minesweeper game board must provide. It respects the Open/Closed Principle since additional game rules can be added by extending

implementations of the interface. The Liskov Substitution Principle is respected as long as implementations preserve expected behavior. The Interface Segregation Principle is respected because it includes only methods relevant to the game logic. It also follows the Dependency Inversion Principle by allowing both the controller and UI to depend on the abstraction.

ICellRenderer interface

Finally, the ICellRenderer interface respects the Single Responsibility Principle by focusing solely on UI rendering actions. It follows the Interface Segregation Principle because it only contains methods needed for updating the cell display and showing messages. It also respects the Dependency Inversion Principle by decoupling the UI from any specific renderer implementation. The remaining SOLID principles apply only indirectly since it is an abstraction.

4. Conclusion

This Minesweeper project demonstrates a clear and modular object-oriented design that effectively separates game logic, data structures, and the user interface. By applying the SOLID principles, the code remains easy to maintain, extend, and test. Each class has a well-defined responsibility, and the use of interfaces ensures flexibility and scalability.