System Architecture Overview

Purpose

Our program allows users to play the game “Minesweeper” on a 10x10 board in the terminal. The user can choose between 10 and 20 mines to be randomly placed on the board for each game. The game supports left and right clicks through a mouse or can be played completely using only keyboard input. The program also gives the user the option to play again at the conclusion of a game. There are 3 separate program builds, one for Linux, another for Mac, and a third for Windows.

Components

This implementation of Minesweeper is composed of several python modules. At the top level, we have two: **src** and **test**. The test module is built using pytest and is intended to provide an automated sequence of unit tests that can be executed to validate that our program functions as expected. The src module contains two main python files, main.py and classes.py. The src module also has a submodule called **tui** which contains the run_tui.py file.

The program is primarily contained within two python files, “classes.py” and “run_tui.py”. The “classes.py” file acts as the backend of the project, containing the necessary functions and classes to organize and handle game logic related tasks. The “run_tui.py” file then acts as the frontend displaying the game information and state to the user in the terminal and processing user input.

We will explain in more detail later in this document, but the main logical organization of our program’s components is split up between the frontend and backend. The frontend is responsible for displaying the user interface, processing user input, passing information to the backend, receiving updates, and then displaying those changes back to the user. The backend is responsible for taking those updates, processing them according to the rules of Minesweeper, and then responding with those results.

These responsibilities can be summarized as follows: the frontend handles the user interface and input handler, while the backend handles the board manager and game logic. The frontend is mostly contained with the run_tui.py file and its Frontend class, while the backend is primarily stored in the classes.py file and its GameManager class.

Outside of the game-specific components and the related Python modules we've created, we also used a tool called *pyinstaller* to create executable versions of our project. These executables are provided in the *dist-<operating system>* folders. The scripts used to generate these files are provided in the main project directory called *make-<operating system>.sh/.bat*. Choose the appropriate script, run it, and new executables will be generated. It will also generate the *build-<operating system>* and *spec-<operating system>* folders. These contain configuration and intermediate build output files that may be useful for debugging purposes.

The project's dependencies are listed below:

- Python 3
- curses (Linux) / windows-curses (Windows)
- Standard python libraries: enum, platform, random
- pyinstaller (to build the project)
- pytest (to run tests using the test module)

Curses is a Python library that allows for the development of text-based user interfaces in the terminal. Instead of printing lines of texts that scroll down, curses offers control of the entire terminal window. It allows cursor movement to any position, color changes, keyboard and mouse input, and updates to parts of the screen without clearing everything. This is what allows the game to have a proper grid interface where the user can click on cells and see updates in place.

Data Flow

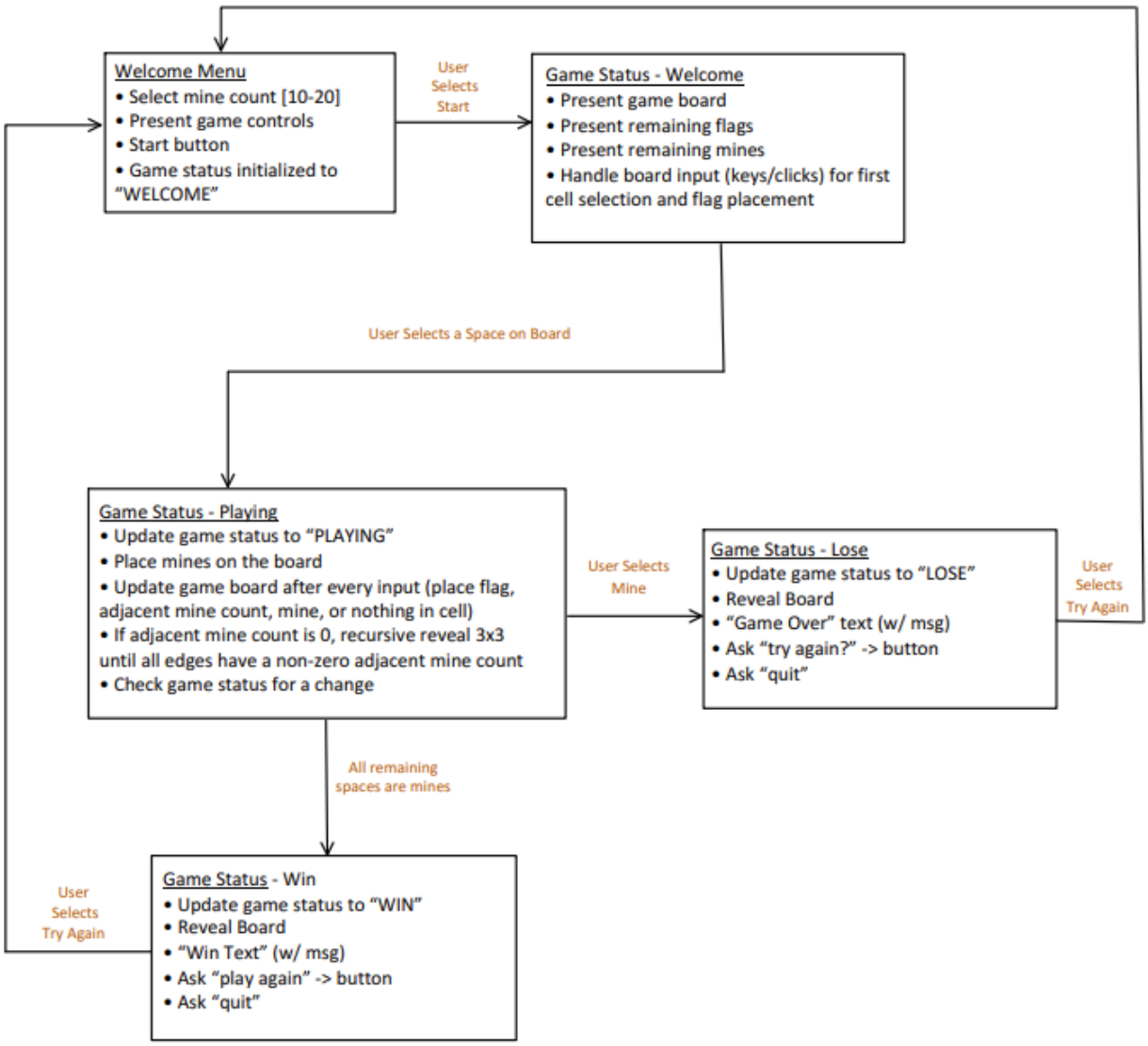

Figure 1

This diagram presents a flowchart of all possible game states within our Minesweeper implementation. It illustrates the program's progression based on player input, beginning at the welcome menu, transitioning into active gameplay upon the first cell selection, and concluding with either a win or loss screen. The flowchart highlights core processes such as mine placement, board updates after each action, and the system's response to different outcomes.

Key Data Structures

The program uses multiple classes to represent the different parts of the game. There are 5 total classes that are used in our implementation. They are **GameManager, Frontend, Cell, CellState,** and **GameStatus,**.

GameManager: GameManager can be thought of as the game's backend. It handles gameplay logic and stores key game-related information, such as the gameplay board represented as a 2D array of rows and columns. This array stores a Cell object for each space in that 2D array (the "grid").

Frontend: Frontend is the class responsible for processing user input and passing the necessary information back to the GameManager. It also retrieves updates from GameManager and updates the user interface to show gameplay updates (e.g. changes to the game board) to the player.

Cell: A **Cell** is used to represent a singular square on the board, and it contains state information about itself, such as the number of squares with adjacent mines or if the Cell itself is flagged. As mentioned previously, the game board is contained within GameManager and is a 2D array of Cells.

CellState: Each Cell also has its own CellState object, which is an enumeration that tracks whether the Cell has no adjacent squares with mines, at least one adjacent square with a mine, or if it has a mine on it.

GameStatus: The GameManager also has a GameStatus object, which is an enumeration that keeps track of the game's status so that the Frontend can handle the visuals accordingly. For example, if the game is still ongoing the GameStatus would be PLAYING. If the player then makes a move which loses the game, the GameStatus would transition to LOSE, and the Frontend would display all the Cells on the board.

The diagram below (Figure 2) outlines each class in the program and its usage.

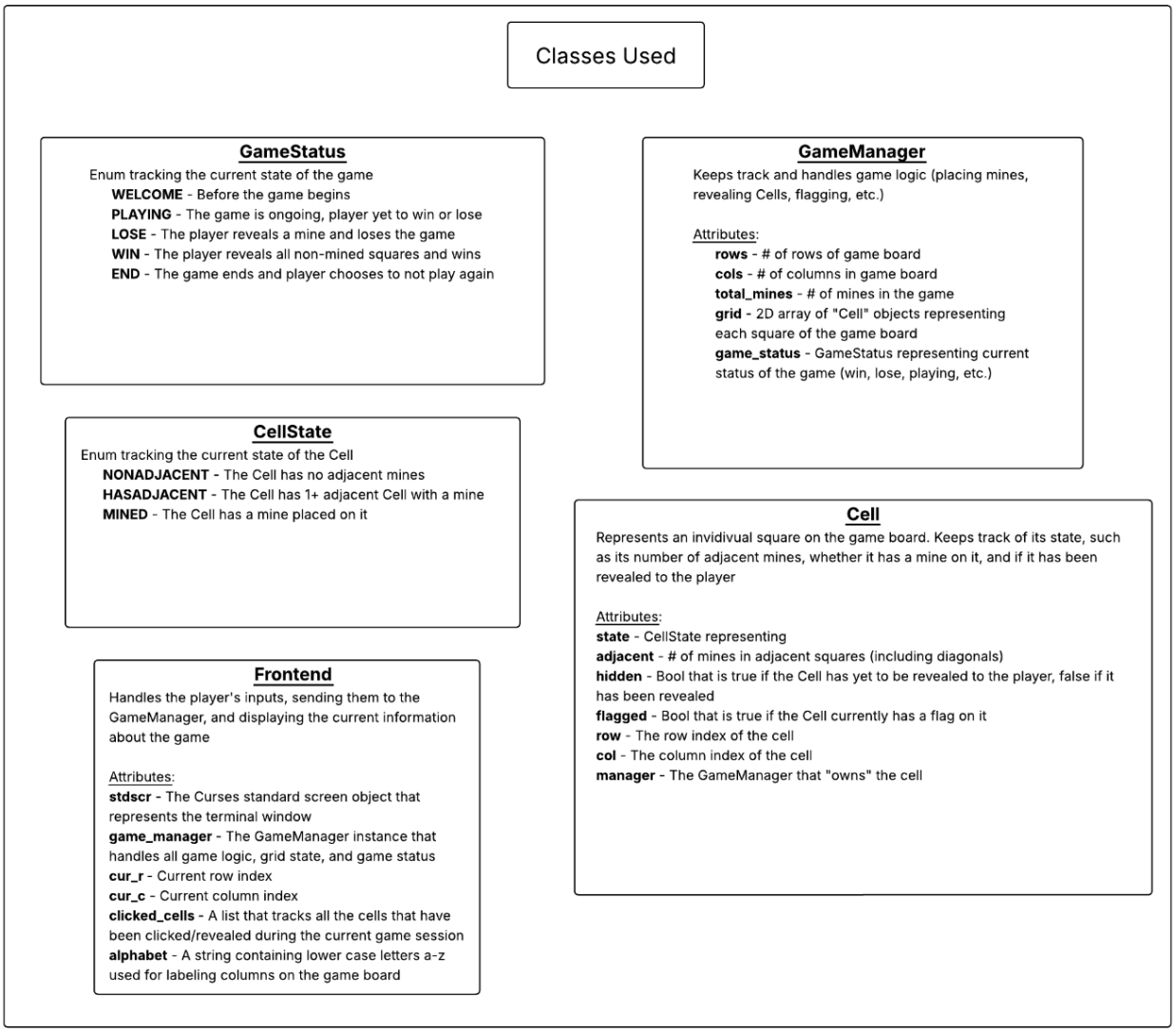

Figure 2

The diagram below (Figure 3) shows the class inheritance hierarchy.

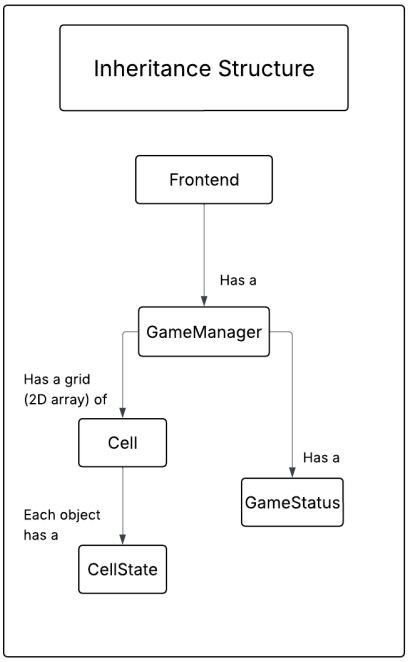

Figure 3

Assumptions

We make the following design assumptions:

Gameplay:

- A fixed 10x10 size grid for gameplay.
- The user will specify a number of mines between 10-20 at the start of the game.

User technical skills:

- The user will know how to download and run the executable from the GitHub repository and build from source if necessary.
- The user will have a valid terminal interface that is properly configured.
- The user will be able to download any dependencies they need.