# Phase 1 - Group EFT

**Introduction:**

For this project, we are making Blockus using Python. The rule will follow the official rule book (https://service.mattel.com/instruction_sheets/BJV44-Eng.pdf). We will use mouse input for all actions in the game (and maybe keyboard input to rotate the pieces). The players can play on the same device. The game can be saved and loaded. The main game screen will consist of the game board and the scoreboard displaying the cumulative score of each player. There are also achievements for players to unlock such as 5 wins in a row, win with a certain colour, the last piece placed is the smallest piece, etc. The AI will use the pieces with the most squares first although the placing does not need to be optimal. The following sections will go further into the user stories, functional/non-functional requirements, and the UML diagram of our game.

**User Stories:**

- As a player, I want to select the number of human players so that I can play with my friends instead of bots.
- As a player, I want to be able to save and load my progress so that I can continue playing from the same spot next time.
- As a player, I want to be able to drag and drop pieces so that I can see how the board will look if I were to place the piece.
- As a player, I want to be able to confirm my placement of a piece so that I can be sure that's the move I want to make.
- As a player, I want to be able to see how the pieces rotate so that I can optimize my piece placements.
- As a player, I want to be able to see each player's score so that I can keep track of who is in the lead.
- As a player, I want to be able to get achievements so that I can feel a sense of accomplishment.
- As a player, I want a clean and clear GUI so that I can easily tell which piece I am placing and the current state of the board.
- As a player, I want a bot that makes good moves so that there is a challenge when I am playing.
- A player finds themselves contemplating 2 potential moves, they would like to assess the state of the board for each option before ending the move. This can be achieved by allowing the player to place a piece and then confirm the placement with an "End Turn" button, or letting them remove the piece in question and try a different one.

**Functional Requirements:**

1. Users will be able to start a new game or load a saved game.
    a. The system will provide a "New Game" and a "Continue Game" buttons
    b. When selecting "New Game," the system will reset the game board, clear all player data, and prompt players with basic start game options.
    c. When selecting "Continue Game," the system will get the last saved game and start from where it was left off.

2. Users will be able to select the number of players participating in the game as well as their colours.
    a. The system should allow users to specify the number of participants before starting a new game and if they are human or bot. -> 2-4 players.
    b. The user can decide how many of the players will be a bot or an actual player (one will always be a local player).
    c. The user(s) will have the possibility of choosing the colour of each player in the game from the available colours. (same colour not possible and one per player)
3. Users will be able to choose their game pieces at the beginning of each turn.
    a. During all gameplay, all available players' pieces are visible on the table(the pieces are visible to all players).
    b. Played pieces will only be on the board and available pieces will be on their respective colour sides, indicating the number of each piece.
    c. Players select the piece with mouse input to select it for placement on the board.
    d. Players can cancel the piece selection and select another piece available.
4. Players will be able to place their chosen game pieces on the board.
    a. The system will allow players to position their selected game pieces on the board during their turns. (save x,y of each piece + possible moves)(use visited?)
    b. The system will show the player where the piece can be played for the player's piece choice and the rotation (90° each rotation).
    c. The system will provide visual indicators to highlight valid positions and prevent overlapping or out-of-bounds placements.
    d. Players can cancel the placement/ preview the placement.
5. The system will enforce the game rules, preventing invalid moves.
    a. The system will validate and restrict players from making illegal moves, such as placing pieces that overlap or extend beyond the board boundaries.
6. The system will determine the winner at the end of the game based on the remaining number of game squares (each piece has a different number of squares 1-5).
    a. The system will calculate the winner of the game based on the number of remaining game pieces for each player.
    b. After every turn, the system will check the number of remaining pieces for each player. Once no more valid moves are possible, the system will compare the remaining piece counts and declare the player with the fewest remaining pieces as the winner.
7. Players will be able to save and load the game state.
    a. The system will provide options to save the game state and load it later for continuation. (if the game is not over)
    b. The system will allow players to save the game state database, capturing the board layout, player turns, and remaining game pieces. Players can then load the saved state from the database to resume the game from where they left off.
8. The system will display the game board and the current state of the game.
    a. The system will provide a visual representation of the game board and relevant game information.

b. The system will render a graphical user interface (GUI) with a grid-based game board, displaying the current positions of game pieces, player names, remaining piece counts, and any other relevant information.

**Non-Functional Requirements:**

1) Reliability: The game should ensure it does not crash or produce errors during playing. Plus, the game should not contain inconsistencies against the rules of Blokus.

2) Response Time(Efficiency of the program): The game should respond to players quickly(The accurate value depends on the types of response, varying from 0.5 to 2.0 seconds.

3) User Interface: The interface of the game should be intuitive. To satisfy players, the interface needs to assist users in easily understanding the game rules and performing actions.

4) Security: The game as a final product should secure and protect its source code, game process, and user data. In addition, the game should be able to defend against attacks or unauthorized tampering from the outside.

5) Data Management: The game should have proper data management functionality, including game scoring and achievement of players. The action of accessing these data for users should be easy.

6) Balance of AI: The AI for games should be moderate, but the level of intelligence could be set differently depending on the level of difficulty.

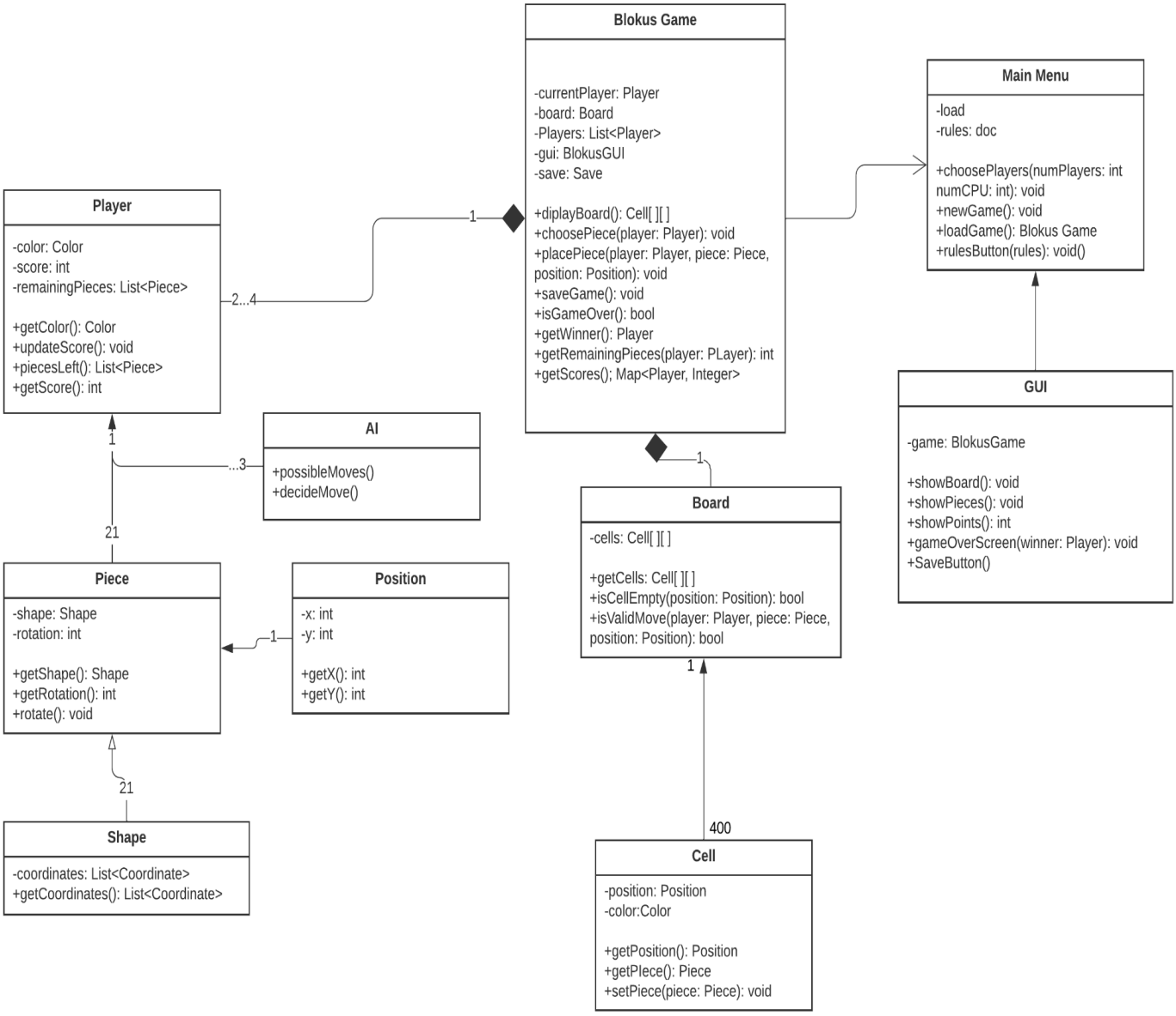
7) Visual and Sound Effects: The game should contain proper visual and sound effects, to increase the immersion of the game.

8) Exception Handling and Fault Tolerance: The game should contain exception-handling methods mechanisms, to handle errors to ensure the stability of the game.

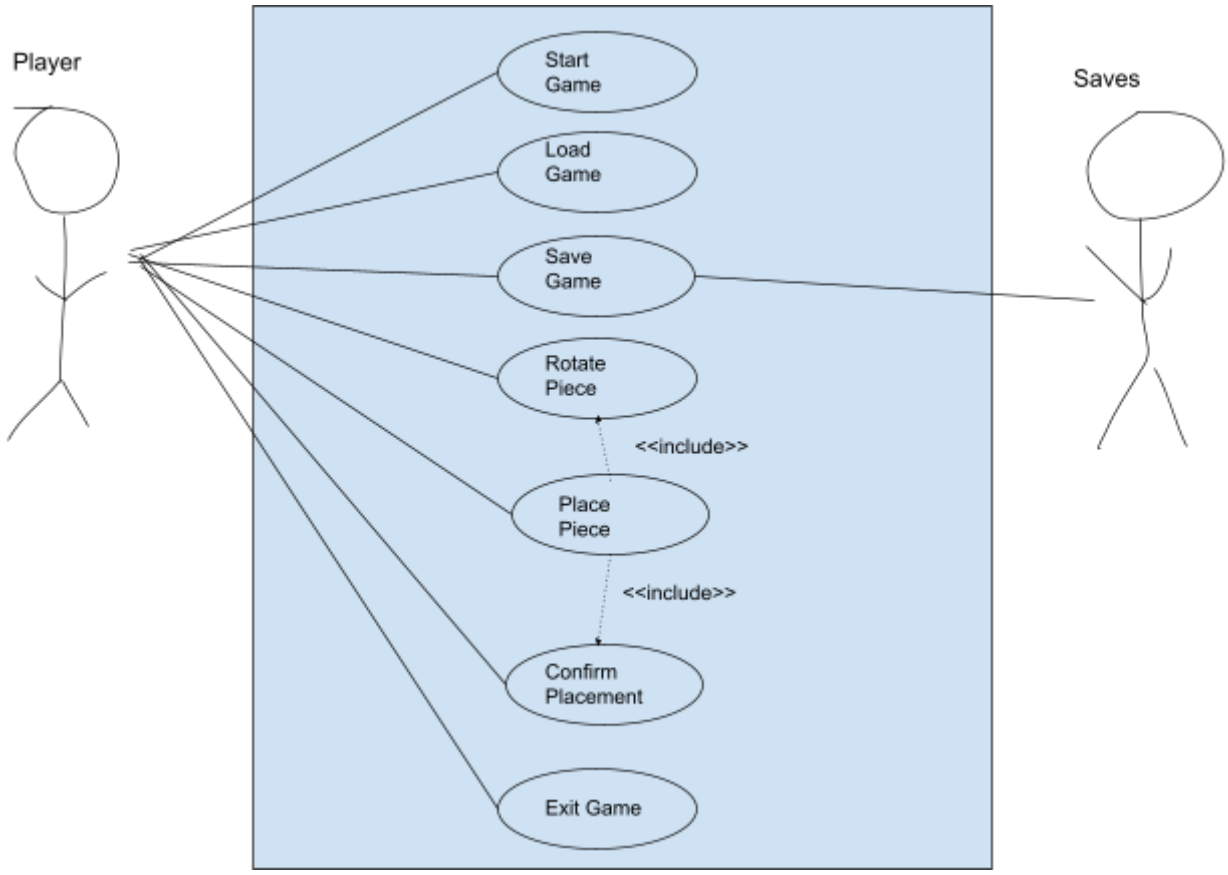
9) Help Documentation: The game should provide appropriate guidance for players to understand contents, objects, gameplay and game variation, etc.

10) User Feedback: The game should provide an appropriate user feedback interface for error prompts and suggestions for the game. This feedback interface informs designers to know bugs and further potential development for the game.

**UML Diagram:**


Blokus Game
-currentPlayer: Player
-board: Board
-Players: List<Player>
-gui: BlokusGUI
-save: Save
+diplayBoard(): Cell[ ][ ]
+choosePiece(player: Player): void
+placePiece(player: Player, piece: Piece, position: Position): void
+saveGame(): void
+isGameOver(): bool
+getWinner(): Player
+getRemainingPieces(player: PLayer): int
+getScores(); Map<Player, Integer>
Main Menu
-load
-rules: doc
+choosePlayers(numPlayers: int numCPU: int): void
+newGame(): void
+loadGame(): Blokus Game
+rulesButton(rules): void()
Player
-color: Color
-score: int
-remainingPieces: List<Piece>
+getColor(): Color
+updateScore(): void
+piecesLeft(): List<Piece>
+getScore(): int
1
2...4
AI
+possibleMoves()
+decideMove()
1
...3
21
GUI
-game: BlokusGame
+showBoard(): void
+showPieces(): void
+showPoints(): int
+gameOverScreen(winner: Player): void
+SaveButton()
1
Board
-cells: Cell[ ][ ]
+getCells: Cell[ ][ ]
+isCellEmpty(position: Position): bool
+isValidMove(player: Player, piece: Piece, position: Position): bool
Piece
-shape: Shape
-rotation: int
+getShape(): Shape
+getRotation(): int
+rotate(): void
Position
-x: int
-y: int
+getX(): int
+getY(): int
1
1
21
1
400
Shape
-coordinates: List<Coordinate>
+getCoordinates(): List<Coordinate>
Cell
-position: Position
-color:Color
+getPosition(): Position
+getPIece(): Piece
+setPiece(piece: Piece): void

**Use-Case UML:**


Player
Start Game
Load Game
Save Game
Rotate Piece
<<include>>
Place Piece
<<include>>
Confirm Placement
Exit Game
Saves