Software Design Document: Clue-Less

*Prepared by **Team GGRZ***

Table of Contents

1. Introduction

1.1 Purpose

The primary objective of this design document is to outline the architecture and class design of the core gameplay mechanics. While comprehensive, the focus is majorly shifted towards the main game process.

1.2 Scope

Certain components, especially those not directly linked to the game's main process, have been either consolidated or omitted for the purpose of clarity and conciseness. Noteworthy mentions include the user management system, lobby management system, and database. While these components are crucial to the overarching system, they have been elaborately discussed in the SRS document and exhibited in the skeletal increment demo.

1.3 Overview

This document will provide a simplified view of the class structures, emphasizing the gameplay logic. The websocket communication, which facilitates interaction between the client and server, is streamlined in our diagrams. Detailed intricacies of this communication protocol have been previously covered in other documents and the skeletal increment demo.

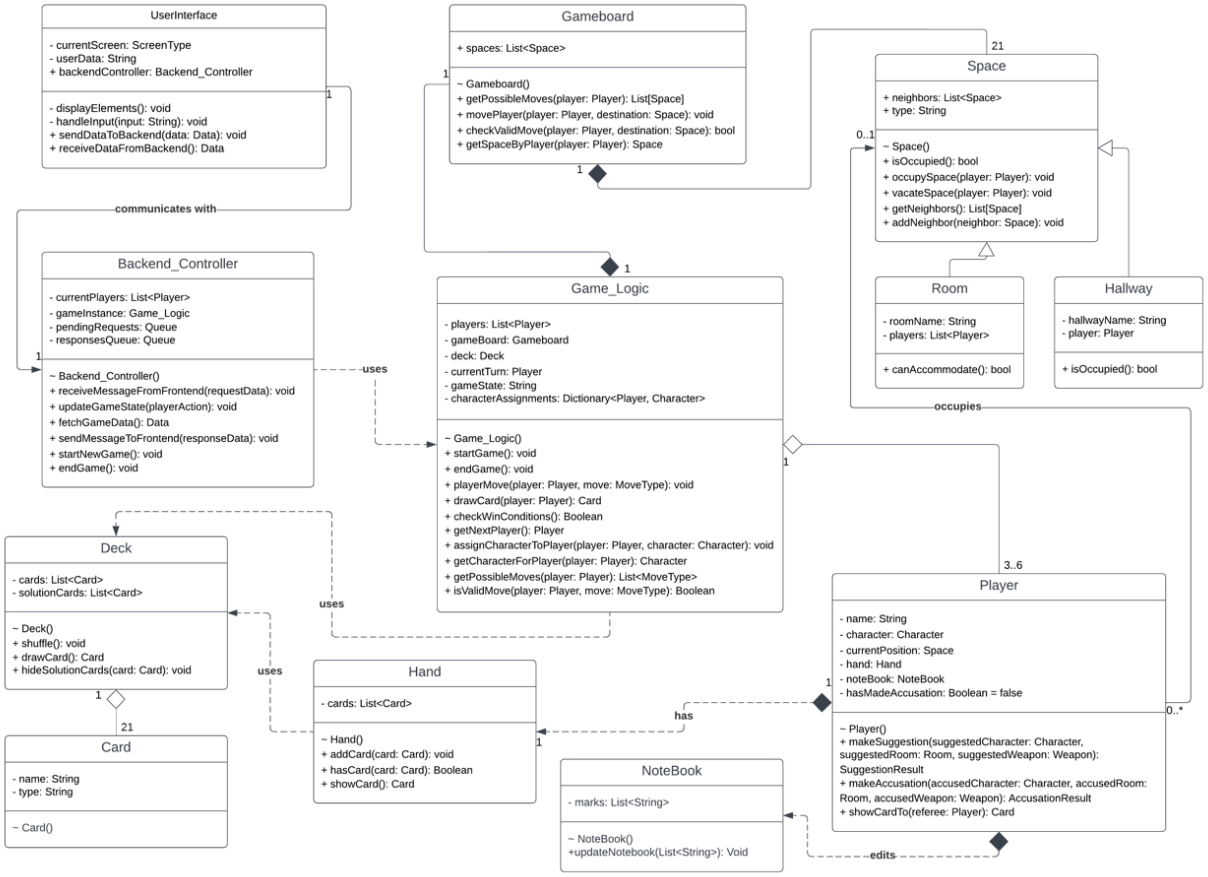
2. Static Design

2.1 Architecture Overview

The game architecture is structured around several key subsystems:

- **UserInterface Subsystem**: Provides the primary interface for players to engage with the game. It collaborates with backend systems to display relevant game data and execute player actions.
- **Backend_Controller Subsystem**: Acts as a bridge between the user interface and the core game logic. It handles player requests, updates game states, and manages data exchanges.
- **Game_Logic Subsystem**: Central to the game mechanics, this subsystem manages gameplay rules, player actions, and game progression. It encompasses components like the game board, players, and game cards.
- **Gameboard Subsystem**: Represents the game's playing area. It includes spaces such as rooms and hallways and defines player movements within the game.
- **Player Management Subsystem**: Manages player attributes, actions, and in-game utilities like the notebook. This subsystem ensures players can interact, make suggestions, and progress within the game.

2.2 Class Diagram



For details, please visit:

Class diagram design in Lucidchart

2.3 Class Specification

Class name: UserInterface

Description: Manages the display and interaction with the player. Communicates with the Backend_Controller to handle game flow and data updates.

Attributes	Type	Description
`currentScreen`	`ScreenType`	The type of screen currently being displayed to the user.
`backendController`	`Backend_Controller`	A reference to the backend controller that handles game logic.
Methods	**Return Type**	**Description**

Attributes	Type	Description
`displayElements()`	`void`	Displays elements on the screen according to the type of `currentScreen`.
`handleInput(input)`	`void`	Processes user input and sends it to the `Backend_Controller`.
`sendDataToBackend(data)`	`void`	Sends processed data to `Backend_Controller` for further handling.
`receiveDataFromBackend()`	`Data`	Receives updates or responses from the `Backend_Controller`.

Class name: Backend_Controller

Description: Central controller for managing game logic, player data, and communication with the UserInterface.

Attributes	Type	Description
`currentPlayers`	`List <Player>`	The list of players currently in the game.
`gameInstance`	`Game_Logic`	The main game logic handler.
`pendingRequest`	`Queue`	Queue for handling pending game requests.
`responseQueue`	`Queue`	Queue for managing responses to be sent to the frontend.
Methods	**Return Type**	**Description**
`Backend_Controller()`	`void`	Constructor to initialize the Backend_Controller.
`receiveMessageFromFrontend(...)`	`void`	Receive and process requests from the frontend.
`updateGameStateUponAction()`	`void`	Update the game state based on game logic.
`fetchGameData()`	`Data`	Get the current game data.
`sendMessageToFrontend(...)`	`void`	Send response or updates to the frontend.
`startNewGame()`	`void`	Begin a new game instance.

Attributes	Type	Description
`endGame()`	`void`	End the current game.

Class name: Game_Logic

Description: Manages the main game logic, rules, and flow of the game.

Attributes	Type	Description
`players`	`List`	List of all players in the current game.
`gameBoard`	`Gameboard`	The main game board instance.
`deck`	`Deck`	Deck of game cards.
`currentTurn`	`Player`	Player whose turn it is currently.
`gameState`	`String`	Current state of the game.
`characterAssignments`	`Dictionary<Player, Character>`	Assignment of characters to players.
Methods	**Return Type**	**Description**
`Game_Logic()`	`void`	Constructor to set up initial game state.
`startGame()`	`void`	Start the game process.
`endGame()`	`void`	End the game and announce results.
`playMove(player, move)`	`void`	Process a player's move.
`drawCard(player)`	`Card`	Allow a player to draw a card.
`checkWinConditions()`	`bool`	Check if win conditions are met.
`getNextPlayer()`	`Player`	Get the next player's turn.

Class name: Gameboard

Description: Represents the game board which consists of various spaces and manages player movements.

Attributes	Type	Description
`spaces`	`List<spaces>`	List of all spaces present on the game board.
Methods	**Return Type**	**Description**

`Gameboard()`	`void`	Constructor to initialize the game board.
`getPossibleMoves(player)`	`List`	Get a list of possible moves for a given player.
`movePlayer(player, destination)`	`void`	Move a player to a specified destination space.
`checkValidMove(player, destination)`	`bool`	Verify if a move is valid for a given player and destination.
`getSpaceByPlayer(player)`	`Space`	Retrieve the space currently occupied by a player.

Class name: Space

Description: Represents a general space on the game board.

Attributes	Type	Description
`neighbors`	`List<Space>`	The adjacent spaces on the game board.
`type`	`String`	The type of the space (either 'Room' or 'Hallway').
Methods	**Return Type**	**Description**
`Space()`	`void`	Constructor for a space.
`isOccupied()`	`bool`	Determines if the space is currently occupied.
`occupySpace(player)`	`void`	Set a player to occupy the space.
`vacateSpace(player)`	`void`	Remove a player from occupying the space.
`getNeighbors()`	`List<Space>`	Retrieve the neighboring spaces.
`addNeighbor(neighbor)`	`void`	Add a neighboring space.

Class name: Room

Description: Represents a room on the game board.

Attributes	Type	Description
`roomName`	`String`	The name of the room.
`players`	`List<Player>`	List of players currently in the room.
Methods	**Return Type**	**Description**
`canAccommodate()`	`bool`	Determines if the room can accommodate more players.

Class name: Hallway

Description: Represents a hallway on the game board.

Attributes	Type	Description
`hallwayName`	`String`	The name of the hallway.
`player`	`Player`	The player currently in the hallway.
Methods	**Return Type**	**Description**
`isOccupied()`	`bool`	Determines if the hallway is currently occupied by a player.

Class name: Player

Description: Represents a player in the game with assigned character, current position, and other relevant details.

Attributes	Type	Description
`name`	`String`	The name of the player.
`character`	`Character`	The character assigned to the player.
`currentPosition`	`Space`	The current space occupied by the player on the game board.
`hand`	`Hand`	The cards held by the player.
`noteBook`	`NoteBook`	The notebook used by the player to keep track of clues.
`hasMadeAccusation`	`Boolean` (default is false)	Flag to indicate if the player has made an accusation.
Methods	**Return Type**	**Description**

Attributes	Type	Description
`Player()`	`void`	Constructor to create a new player instance.
`makeSuggestion(suggested...`	`SuggestionResult`	Allows the player to make a suggestion.
`makeAccusation(accused...`	`AccusationResult`	Allows the player to make an accusation.
`showCardTo(referee)`	`Card`	Show a card to a referee player during a suggestion.

Class name: Deck

Description: Manages the collection of game cards.

Attributes	Type	Description
`cards`	`List<Card>`	A list of cards in the deck.
`solutionCards`	`List<Card>`	A list of cards which is the solution.
Methods	**Return Type**	**Description**
`Deck()`	`void`	Constructor to initialize the deck.
`shuffle()`	`void`	Shuffle the cards in the deck.
`drawCard()`	`Card`	Draw a card from the top of the deck.
`hideSolutionCards(card)`	`void`	Put one of each typed cards into a secret pile, the solution.

Class name: Card

Description: Represents a game card with a specific name and type.

Attributes	Type	Description
`name`	`String`	The name of the card.
`type`	`String`	The type of the card (e.g., Weapon, Character, Room).
Methods	**Return Type**	**Description**

Attributes	Type	Description
`Card()`	`void`	Constructor to create a new card instance.

Class name: Hand

Description: Represents a player's hand containing a set of game cards.

Attributes	Type	Description
`cards`	`List`	List of cards in the player's hand.
Methods	**Return Type**	**Description**
`Hand()`	`void`	Constructor to initialize a hand.
`addCard(card)`	`void`	Add a card to the player's hand.
`hasCard(card)`	`Boolean`	Check if the hand contains a specific card.
`showCard()`	`Card`	Display a card from the player's hand.

Class name: NoteBook

Description: Represents a player's notebook to keep track of clues and findings.

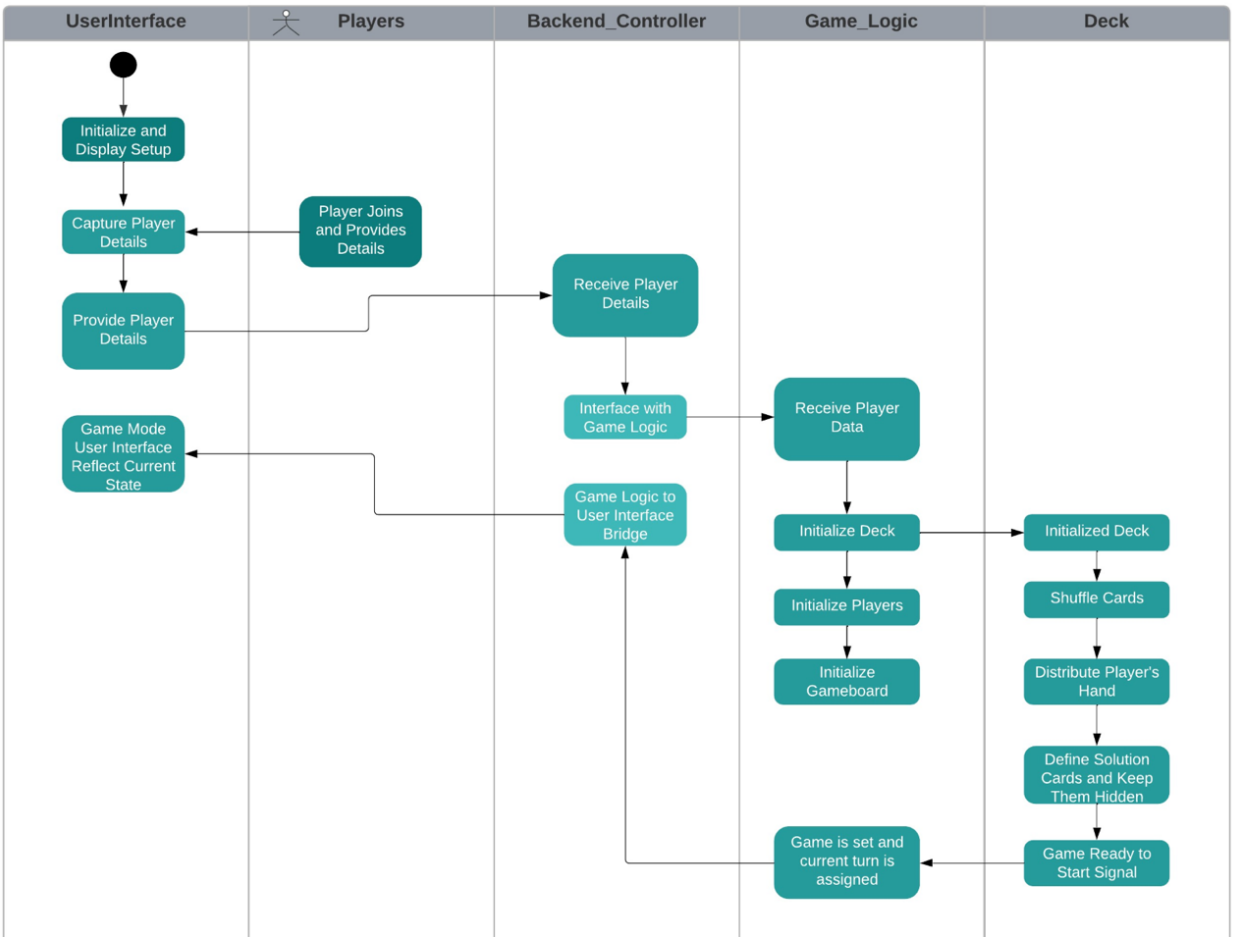
Attributes	Type	Description
`marks`	`List`	List of marked clues or findings.
Methods	**Return Type**	**Description**
`NoteBook()`	`void`	Constructor to initialize a new notebook.
`updateNotebook(list)`	`Void`	Updates the notebook with new marks or clues.

3. Dynamic Design

3.1 Starting a New Game

- The **UserInterface** initializes and displays the game setup elements.

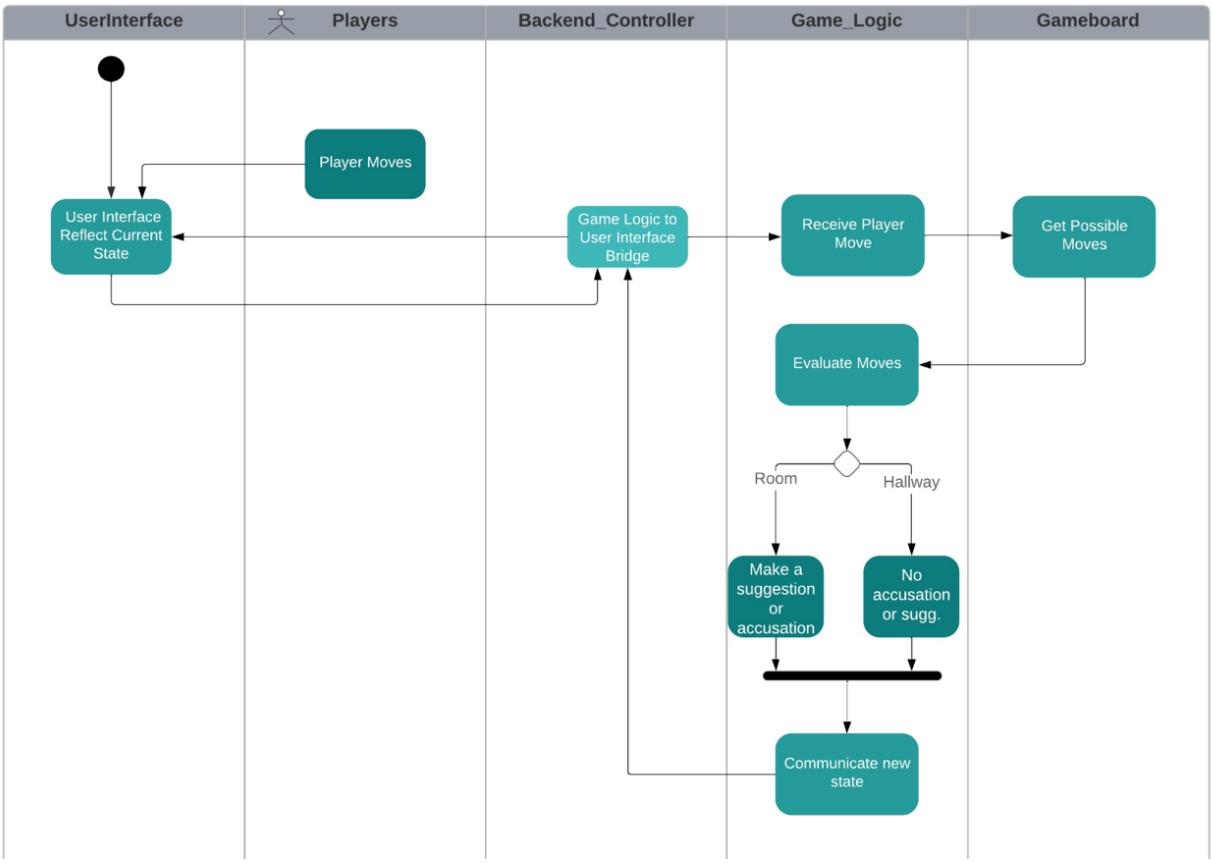
- Players join the game. Their details are captured by the **UserInterface** and sent to the **Backend_Controller**.
- The **Backend_Controller** communicates with **Game_Logic** to initialize the **Gameboard**, **Deck**, and **Player** objects.
- The **Deck** shuffles the cards and distributes them to players' **Hand** and keeps the **solutionCards** hidden.
- The game state is set, and the **currentTurn** is assigned to the first player.



3.2 Player's Turn

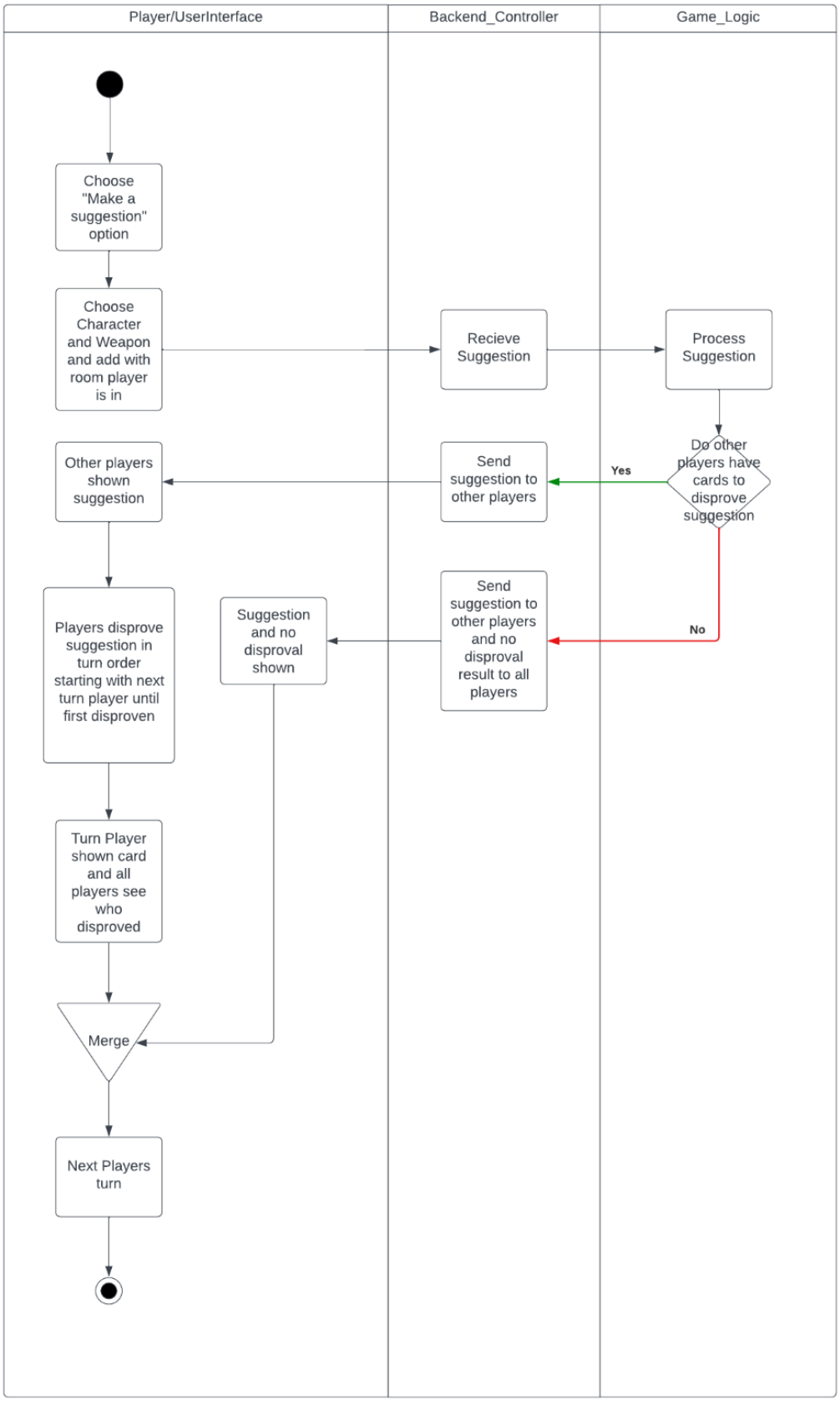
- The **UserInterface** displays the current player's turn.
- The player chooses to move using the **UserInterface**. This action is communicated to the **Backend_Controller**.

- The **Game_Logic** checks with **Gameboard** using **getPossibleMoves** to see where the player can move.
- The player moves to a chosen **Space** (either a **Room** or **Hallway**).
- If a player enters a **Room**, they may make a suggestion or accusation.



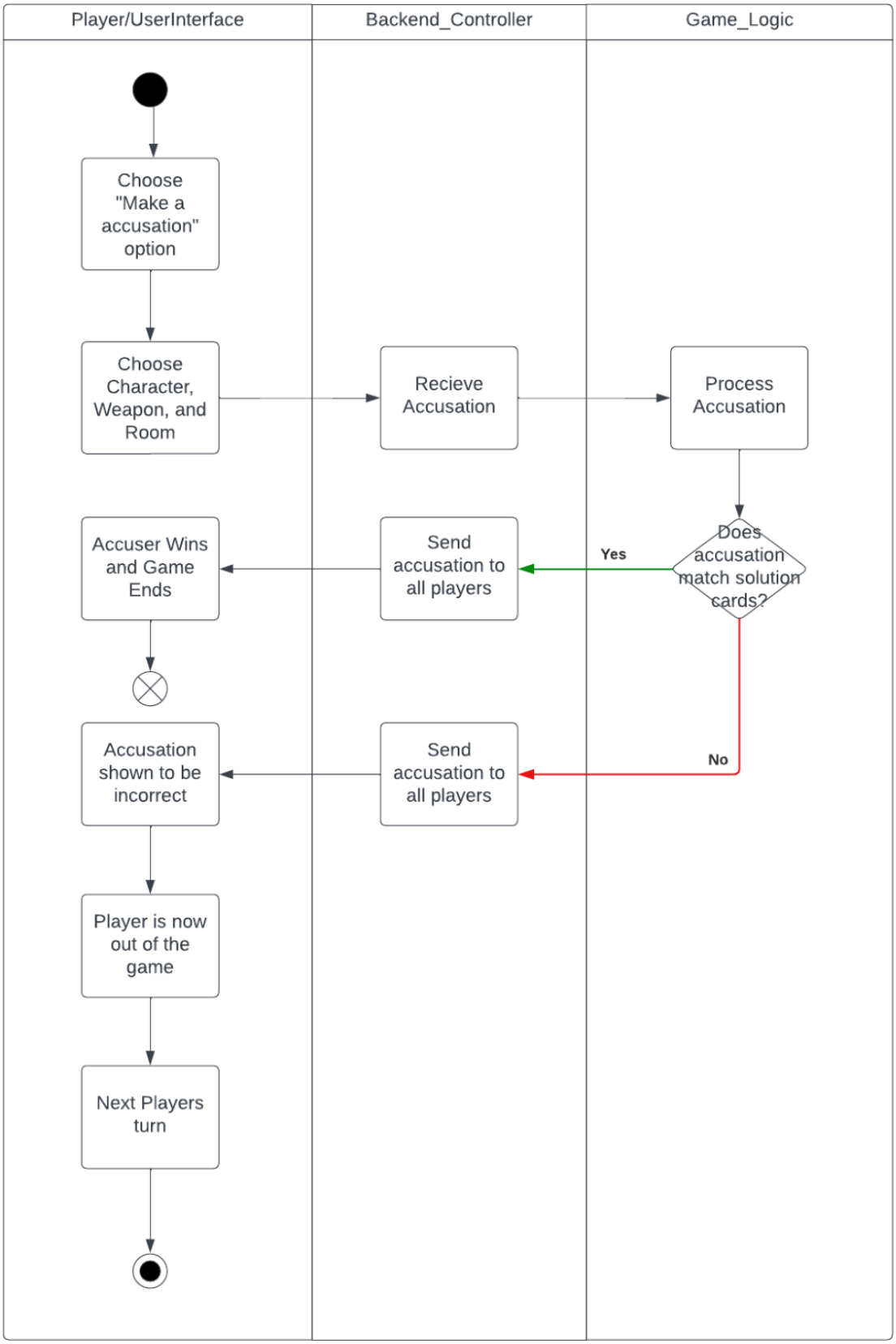
3.3 Making a Suggestion

- The player chooses characters and weapons to suggest a possible solution via the **UserInterface**.
- The **Backend_Controller** receives the suggestion and passes it to **Game_Logic**.
- The **Game_Logic** processes the suggestion. If another player has a card that disproves the suggestion, that card is shown to the suggesting player.
- The game continues to the next player's turn.


Player/UserInterface
Backend_Controller
Game_Logic
Choose "Make a suggestion" option
Choose Character and Weapon and add with room player is in
Recieve Suggestion
Process Suggestion
Do other players have cards to disprove suggestion
Yes
Send suggestion to other players
Other players shown suggestion
No
Send suggestion to other players and no disproval result to all players
Suggestion and no disproval shown
Players disprove suggestion in turn order starting with next turn player until first disproven
Turn Player shown card and all players see who disproved
Merge
Next Players turn

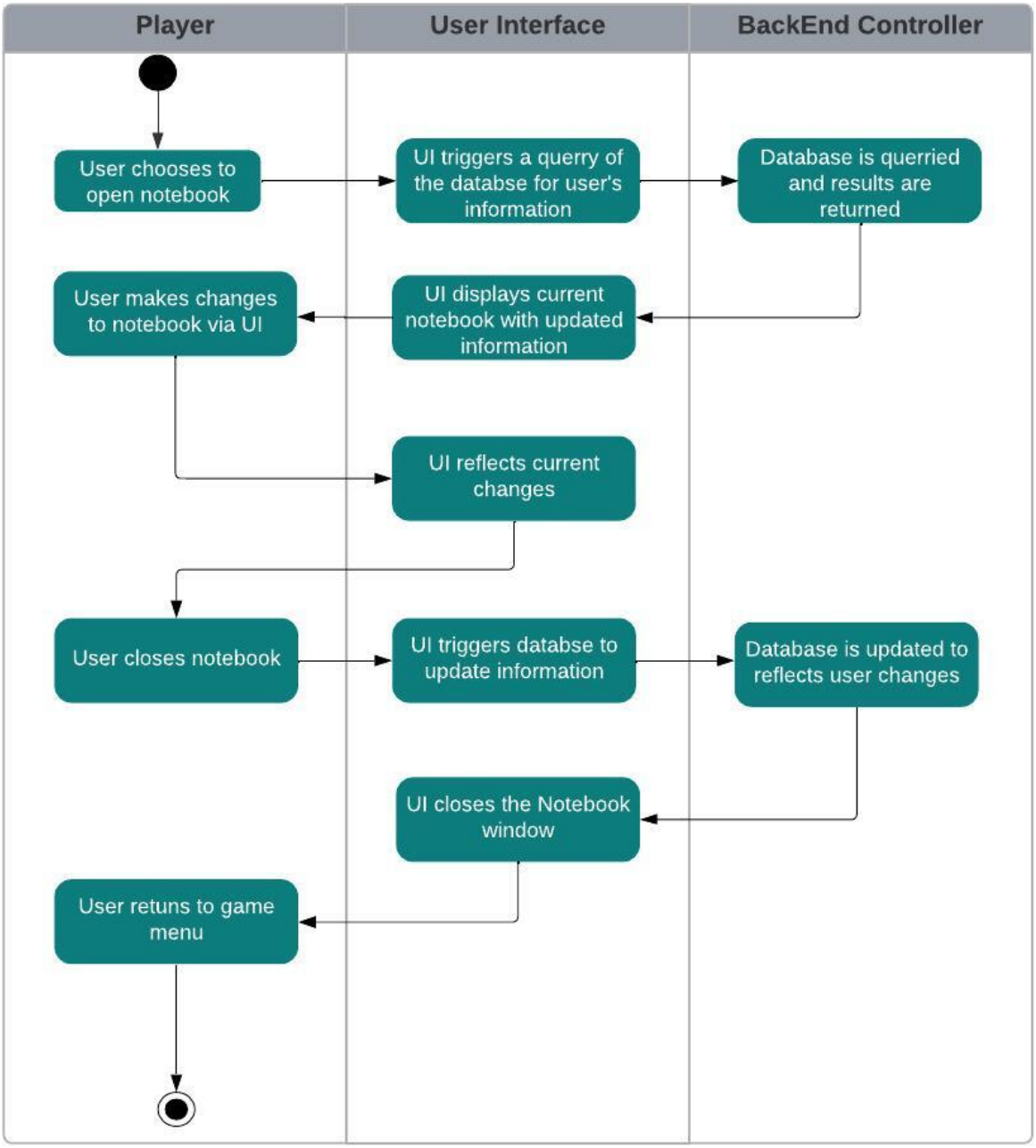
3.4 Making an Accusation:

- If a player feels confident, they make an accusation through the **UserInterface**.
- The **Backend_Controller** communicates this to **Game_Logic**.
- The **Game_Logic** checks the accusation against the **solutionCards** in the **Deck**.
- If the accusation is correct, the player wins and the game ends. Otherwise, the player is out of the game.


Player/UserInterface
Backend_Controller
Game_Logic
Choose "Make a accusation" option
Choose Character, Weapon, and Room
Recieve Accusation
Process Accusation
Does accusation match solution cards?
Yes
Send accusation to all players
Accuser Wins and Game Ends
No
Send accusation to all players
Accusation shown to be incorrect
Player is now out of the game
Next Players turn

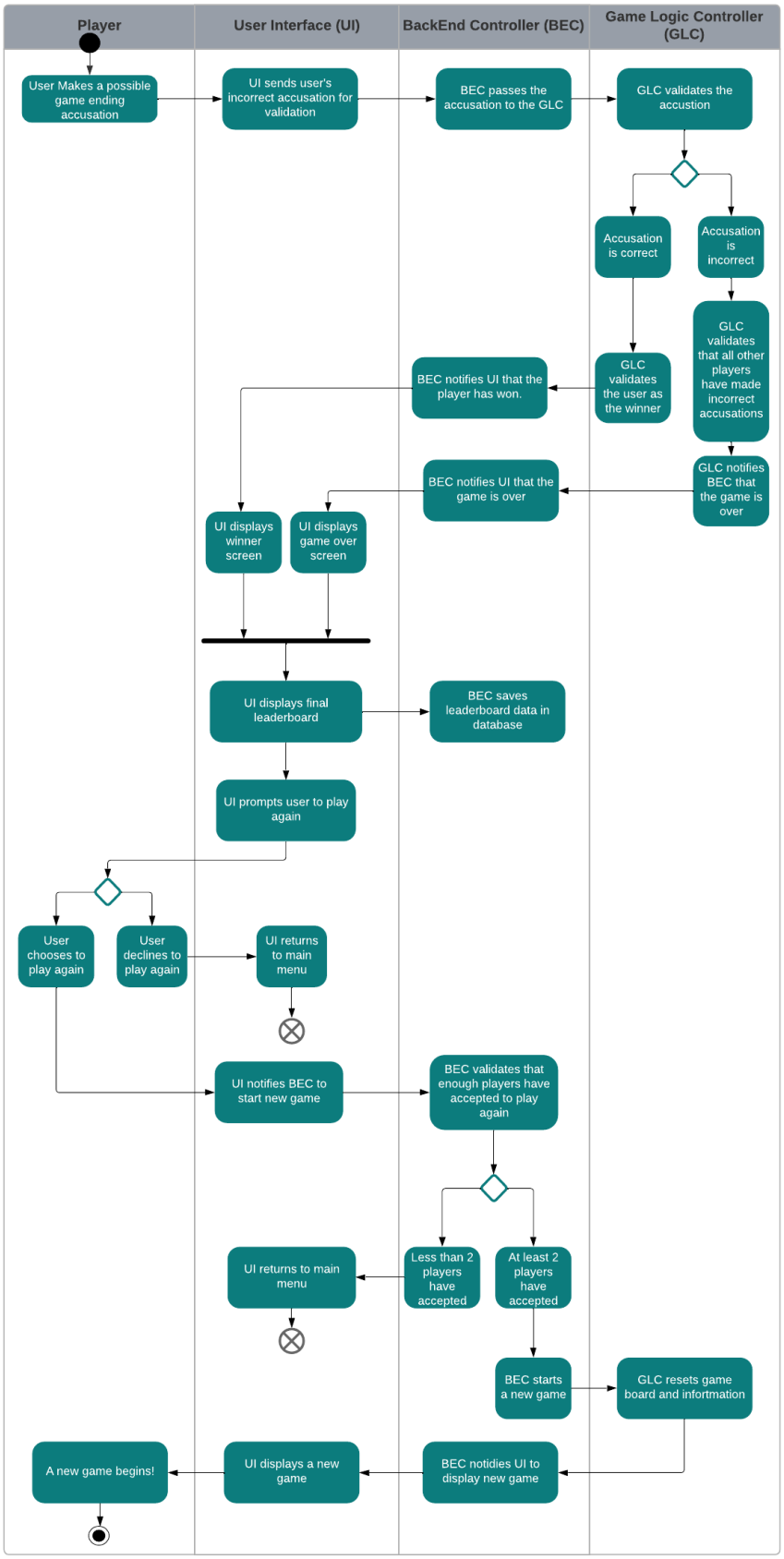
3.5 Keeping Track

- Players keep track of the clues and seen cards in their **NoteBook**.
- They can update their **NoteBook** after seeing a card or hearing a suggestion/accusation.
- The **NoteBook** updates with marks or notes based on player inputs.



3.6 Game Ending Scenarios

- A player makes a correct accusation.
- All players have made incorrect accusations, leading to an end without a winner.
- The **Game_Logic** communicates the end of the game to the **Backend_Controller**, which in turn notifies the **UserInterface**.
- The **UserInterface** displays the game results.


Player
User Interface (UI)
BackEnd Controller (BEC)
Game Logic Controller (GLC)
User Makes a possible game ending accusation
UI sends user's incorrect accusation for validation
BEC passes the accusation to the GLC
GLC validates the accustion
Accusation is correct
Accusation is incorrect
GLC validates that all other players have made incorrect accusations
GLC validates the user as the winner
BEC notifies UI that the player has won.
GLC notifies BEC that the game is over
BEC notifies UI that the game is over
UI displays winner screen
UI displays game over screen
UI displays final leaderboard
BEC saves leaderboard data in database
UI prompts user to play again
User chooses to play again
User declines to play again
UI returns to main menu
UI notifies BEC to start new game
BEC validates that enough players have accepted to play again
Less than 2 players have accepted
At least 2 players have accepted
UI returns to main menu
BEC starts a new game
GLC resets game board and infortmation
A new game begins!
UI displays a new game
BEC notidies UI to display new game