Chapter 1

Class Index

1.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 2

Project Synopsis

2.1 Introduction

The Planning Poker Game is a collaborative estimation tool designed for Agile teams to facilitate accurate project task estimation. This project aims to develop a user-friendly Planning Poker application using Python and Tkinter.

2.2 Objectives

- Create an interactive and intuitive user interface.
- Implement the Planning Poker algorithm for estimating project tasks.
- Allow team members to join sessions and participate in the estimation process.
- Provide real-time updates and notifications during the estimation process.

2.3 Motivation for Developing Planning Poker Game in Python

Agile methodologies have become the cornerstone of modern software development practices, emphasizing adaptability, collaboration, and iterative progress. One crucial aspect of Agile project management is the accurate estimation of task complexity and effort. Traditional estimation methods often fall short in Agile environments due to their rigid nature and lack of collaboration.

The Planning Poker Game serves as a response to the challenges faced by Agile teams in task estimation. By developing this tool in Python, we aim to harness the flexibility and ease of development offered by the language. Python's simplicity, readability, and extensive support for GUI development through frameworks like Tkinter make it an ideal choice for creating an intuitive and user-friendly Planning Poker application.

The motivation for this project lies in addressing the following key considerations:

- **Enhancing Collaboration:** Planning Poker encourages active participation and discussion among team members during the estimation process. Python's ease of use and readability contribute to creating an accessible platform for collaborative decision-making.
- **Increasing Transparency:** Transparency is a core value in Agile methodologies. The Planning Poker Game, developed in Python, will provide real-time updates and consensus information, fostering transparency within the team.
- **Accurate Task Estimation:** Python's support for algorithm development aligns well with the requirements of the Planning Poker algorithm. The aim is to implement a robust estimation mechanism that ensures accurate and reliable task assessments.
- **User-Friendly Interface:** Python's Tkinter framework allows for the creation of a user-friendly and visually appealing interface. This aspect is crucial to encourage adoption and make the estimation process accessible to all team members.
- **Contribution to Agile Success:** The Planning Poker Game, developed in Python, seeks to contribute to the overall success of Agile development methodologies. By providing a reliable estimation tool, the project aims to enhance the efficiency of Agile teams in delivering high-quality software.

In summary, the motivation for developing the Planning Poker Game in Python is rooted in the need for a flexible, collaborative, and transparent tool to address the unique challenges posed by Agile project management.

2.4 Scope of the Project

The Planning Poker Game aims to address the challenges faced by Agile teams in task estimation by providing a collaborative and real-time estimation tool. The scope of the project encompasses the following key features:

2.4.1 Task Estimation

The primary focus of the Planning Poker Game is to implement the Planning Poker algorithm for collaborative task estimation. This involves creating an interactive and user-friendly interface where team members can contribute to the estimation process. The algorithm will allow team members to assign complexity points to different tasks based on consensus, leveraging the collective intelligence of the team for more accurate estimations.

2.4.1.1 User Stories

The system will support the creation and management of user stories, allowing Agile teams to break down tasks into manageable units. Each user story will be subject to estimation during Planning Poker sessions.

2.4.1.2 Estimation Scale

The game will incorporate a flexible estimation scale, allowing teams to use Fibonacci numbers, T-shirt sizes, or other popular estimation scales. This adaptability ensures that the Planning Poker Game can accommodate the preferences of diverse Agile teams.

2.4.1.3 Historical Data

The system will maintain a historical record of task estimations, providing valuable insights into the team's estimation accuracy over time. This data can be used for retrospective analysis and continuous improvement.

2.4.1.4 Live Consensus Tracking

The system will dynamically track the progress of the estimation session, displaying live updates on the consensus-building process. This feature aims to reduce the time taken for estimation by allowing team members to see how close they are to reaching an agreement.

2.4.1.5 User Feedback

To further improve the user experience, the system will provide instant feedback to users about their contributions to the estimation process. This feedback mechanism aims to encourage active participation and ensure that all team members feel heard and valued.

2.4.2 Future Enhancements

While the initial scope focuses on task estimation and real-time updates, future enhancements may include additional features such as integrations with project management tools, advanced reporting capabilities, and support for distributed teams.

By implementing these features, the Planning Poker Game aspires to become a comprehensive and indispensable tool for Agile teams engaged in collaborative project planning and estimation.

2.5 Technology Stack

The Planning Poker Game will be developed using the following technology stack:

- **Programming Language:** Python
- **GUI Framework:** Tkinter

2.6 Significance of the Project

The Planning Poker Game aims to enhance the efficiency of Agile teams by providing a reliable and user-friendly tool for project task estimation. It encourages collaboration, transparency, and accurate estimation, contributing to the success of Agile development methodologies.

Chapter 3

Project Presentation

3.1 Output Interfaces

3.1.1 Main Menu

Here I have made a menu page from where users can navigate to different features. This includes taking players' count and names selecting rules and then entering features in JSON format after that voting process and then saving and loading progress.

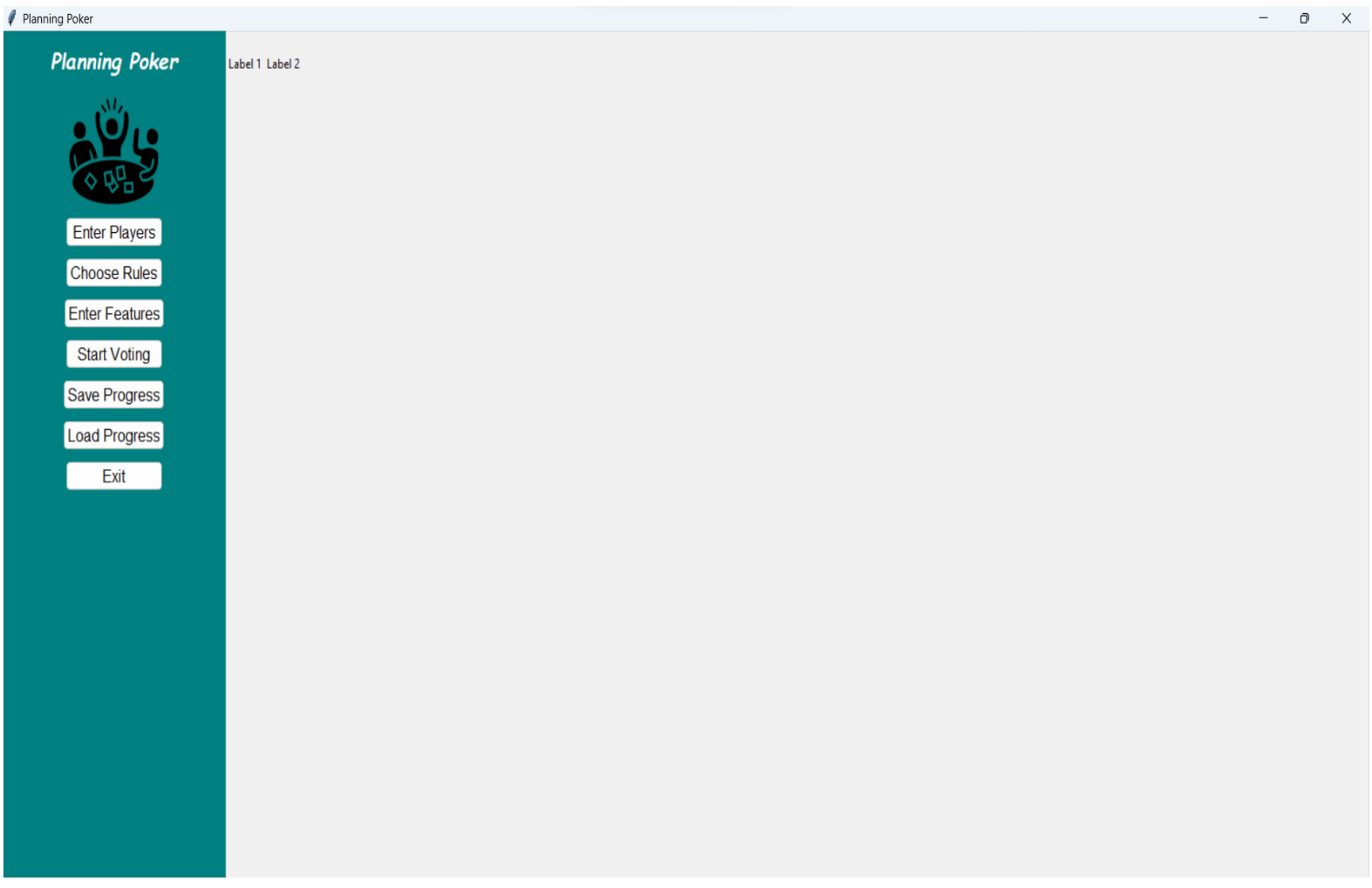

Figure 3.1 Main Page Showing menu of Planning Poker

3.1.2 Enter Players

This interface has the feature of choosing players that involves taking input for the number of players and their names. Users can input how many players will be participating and assign names to each player. This information is likely essential for the subsequent features, such as voting and tracking progress.

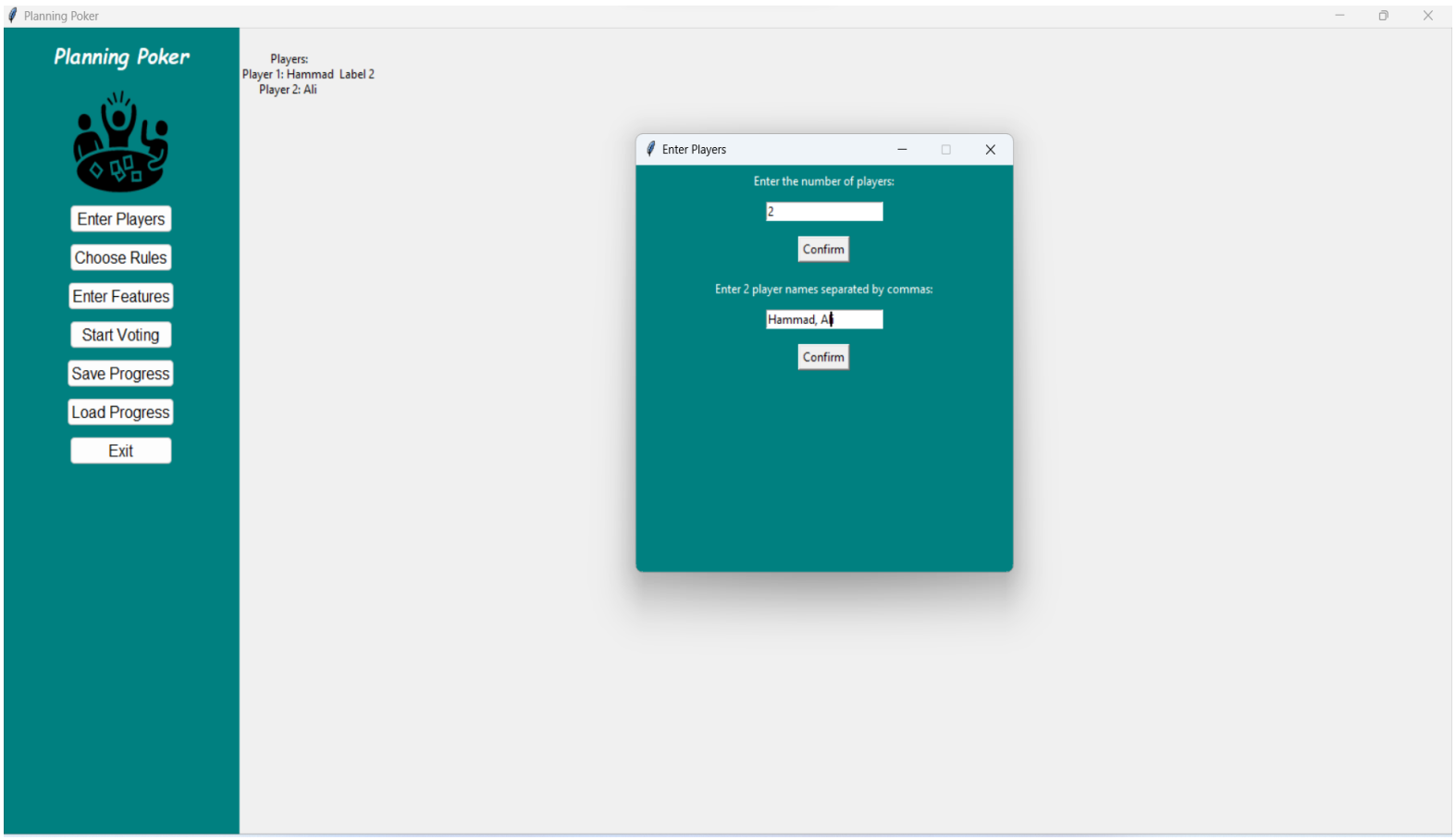

Figure 3.2 Choose Players Count and Names

3.1.3 Choose Rules

Users can select rules for the game or application. This could involve choosing specific settings, options, or parameters that govern how the program operates. Rules can significantly impact the user experience, so providing options for customization is a valuable feature.

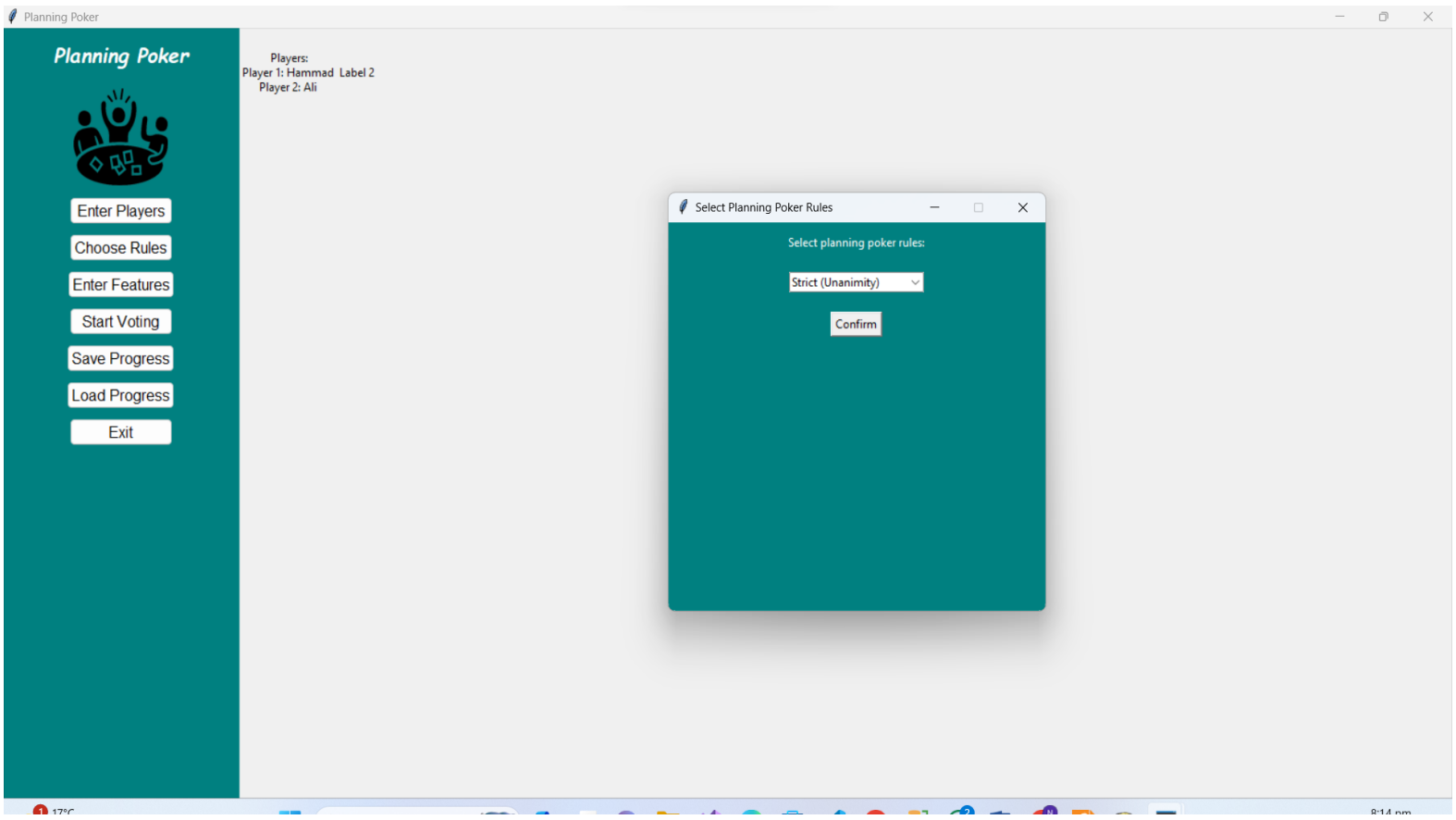

Figure 3.3 Choose Roles

3.1.4 Enter Features

This part involves allowing users to input features or settings in JSON (JavaScript Object Notation) format. JSON is a lightweight data interchange format that is easy for both humans and machines to read and write. Users might be entering specific configurations or preferences for the game or application in a structured format.

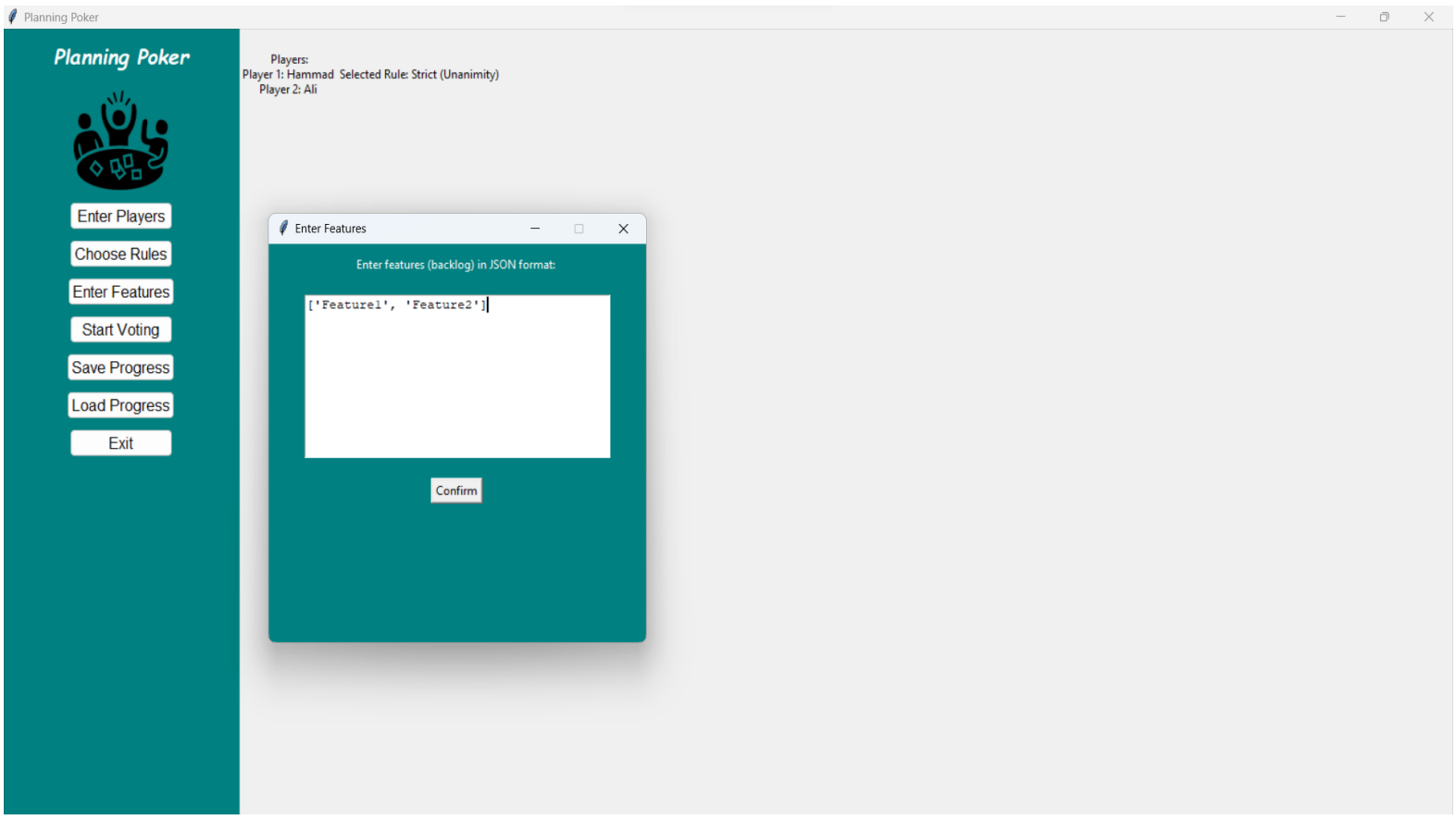

Figure 3.4 Enter Features

3.1.5 Voting Process

This could be a core gameplay or decision-making aspect. Users may participate in a voting process, potentially related to the rules selected or other in-game decisions. The voting process adds an interactive and collaborative element to the application, involving the players in shaping the experience.

3.1.5.1 First Players Voting

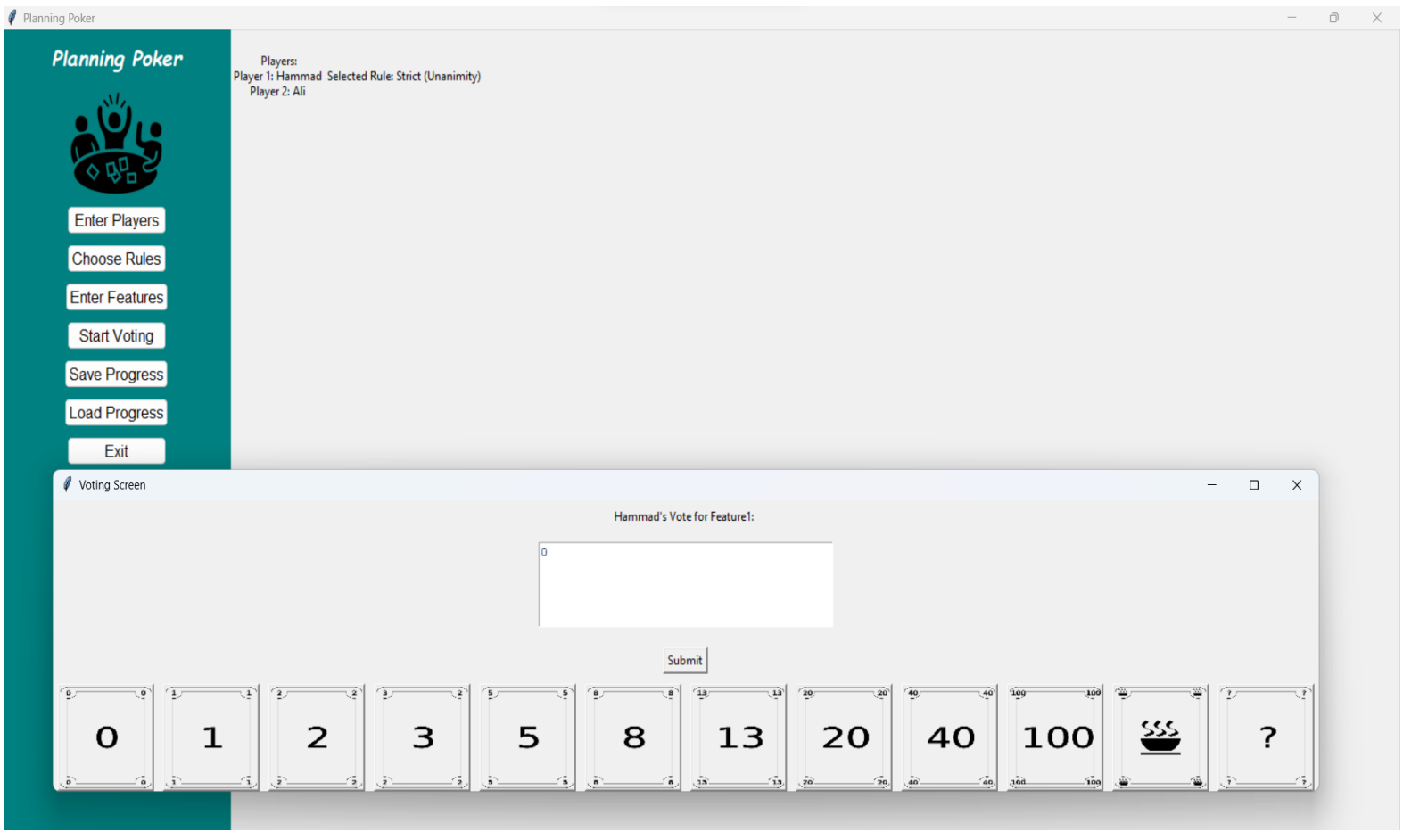

Figure 3.5 Voting of First Player for Feature1

3.1.5.2 Second Players Voting

Now the Second player will give a vote to feature1 and this process goes on.

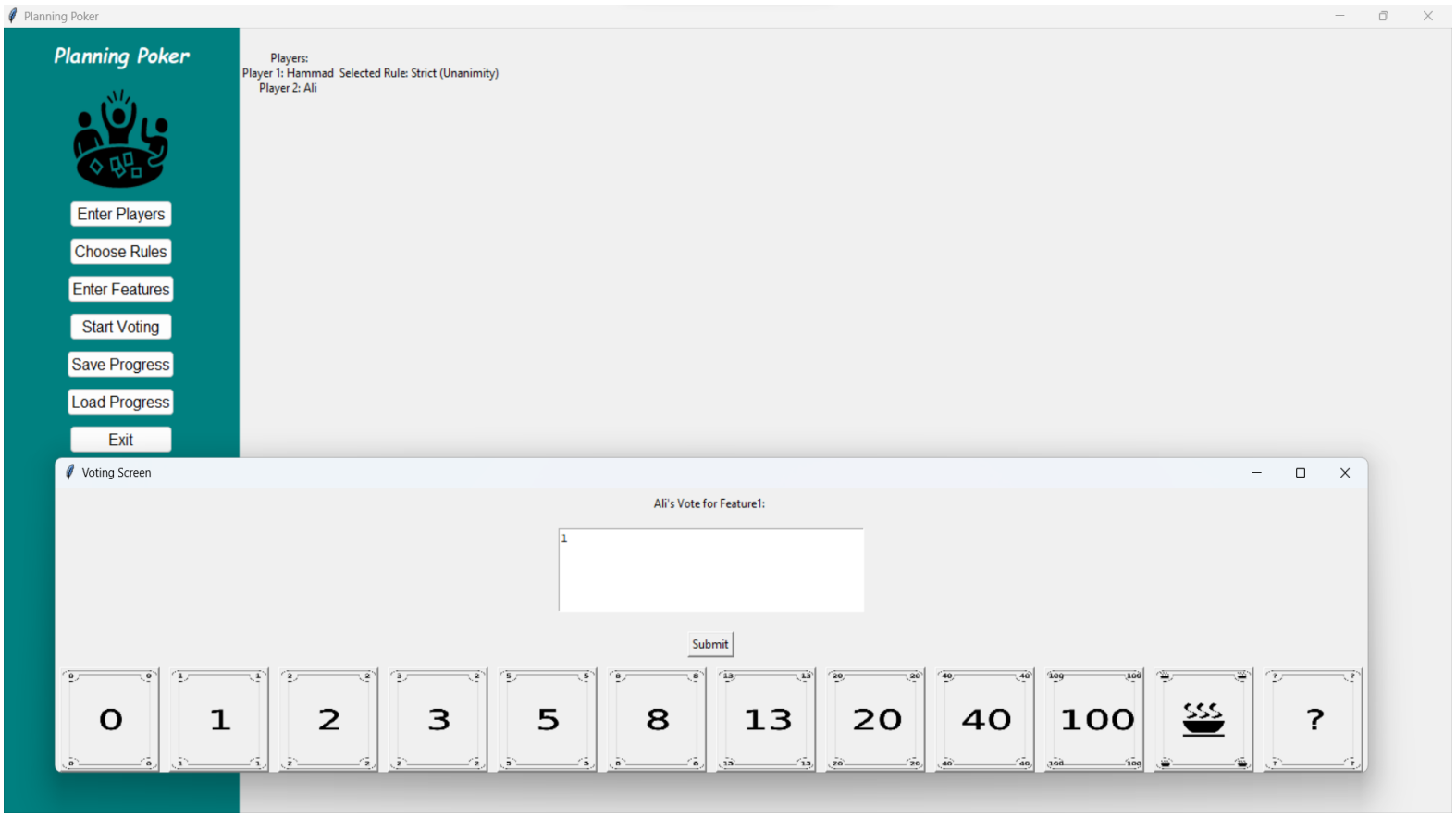

Figure 3.6 Voting of Second Player for Feature1

3.1.5.3 Confirmation Screen

This process goes on until all features are voted on. The below snaps show that voting for all features is done.

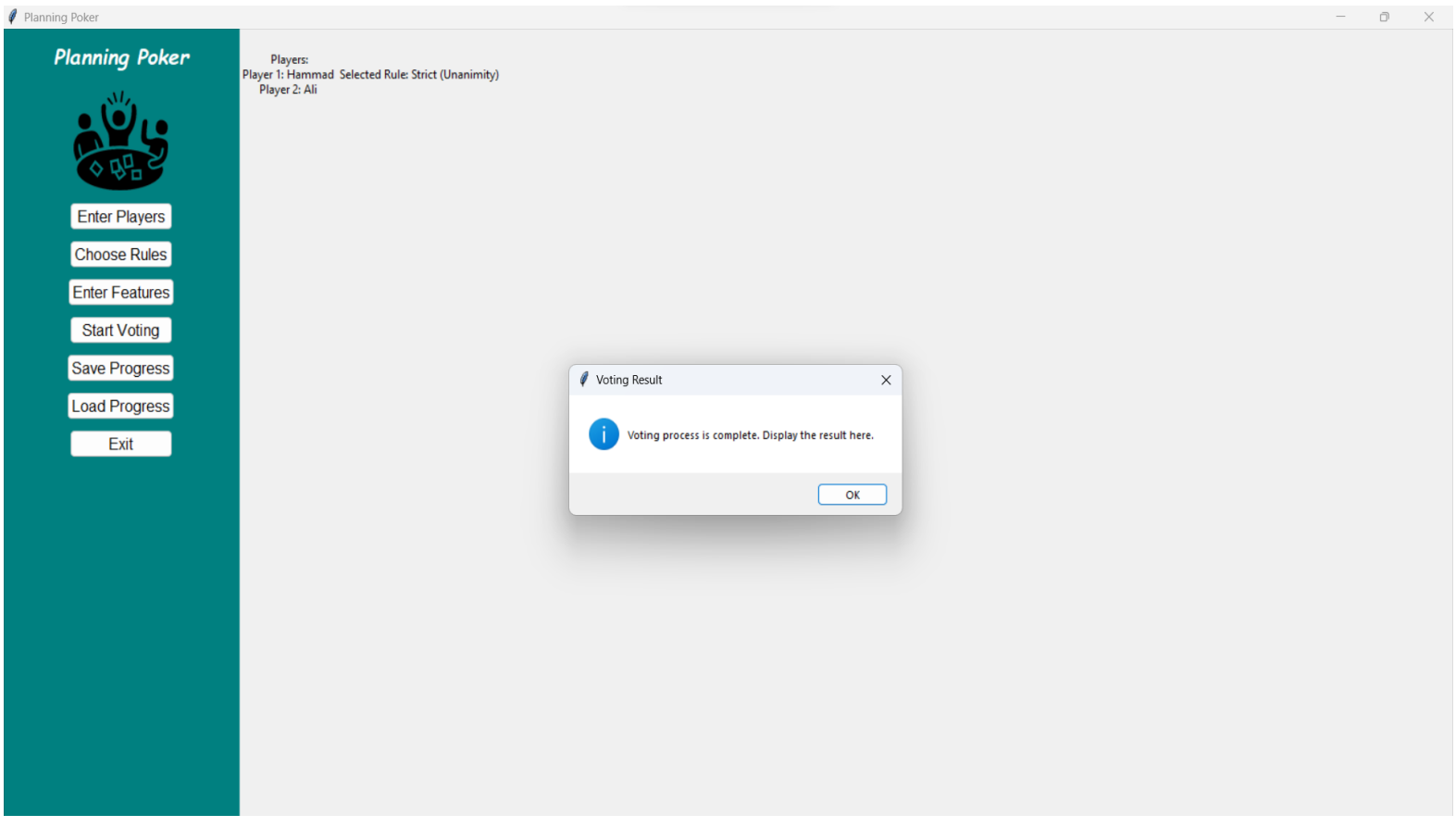

Figure 3.7 Confirmation Screen

3.1.6 Saving and Loading Progress

This feature allows users to save their current state or progress in the application. It could include saving game configurations, player names, rule settings, and any other relevant data. Loading progress allows users to resume a previous session, providing continuity and a sense of persistence.

3.1.6.1 Saving Progress

This includes saving the progress of planning Poker game that has game configurations, player names, rule settings, and any other relevant data.

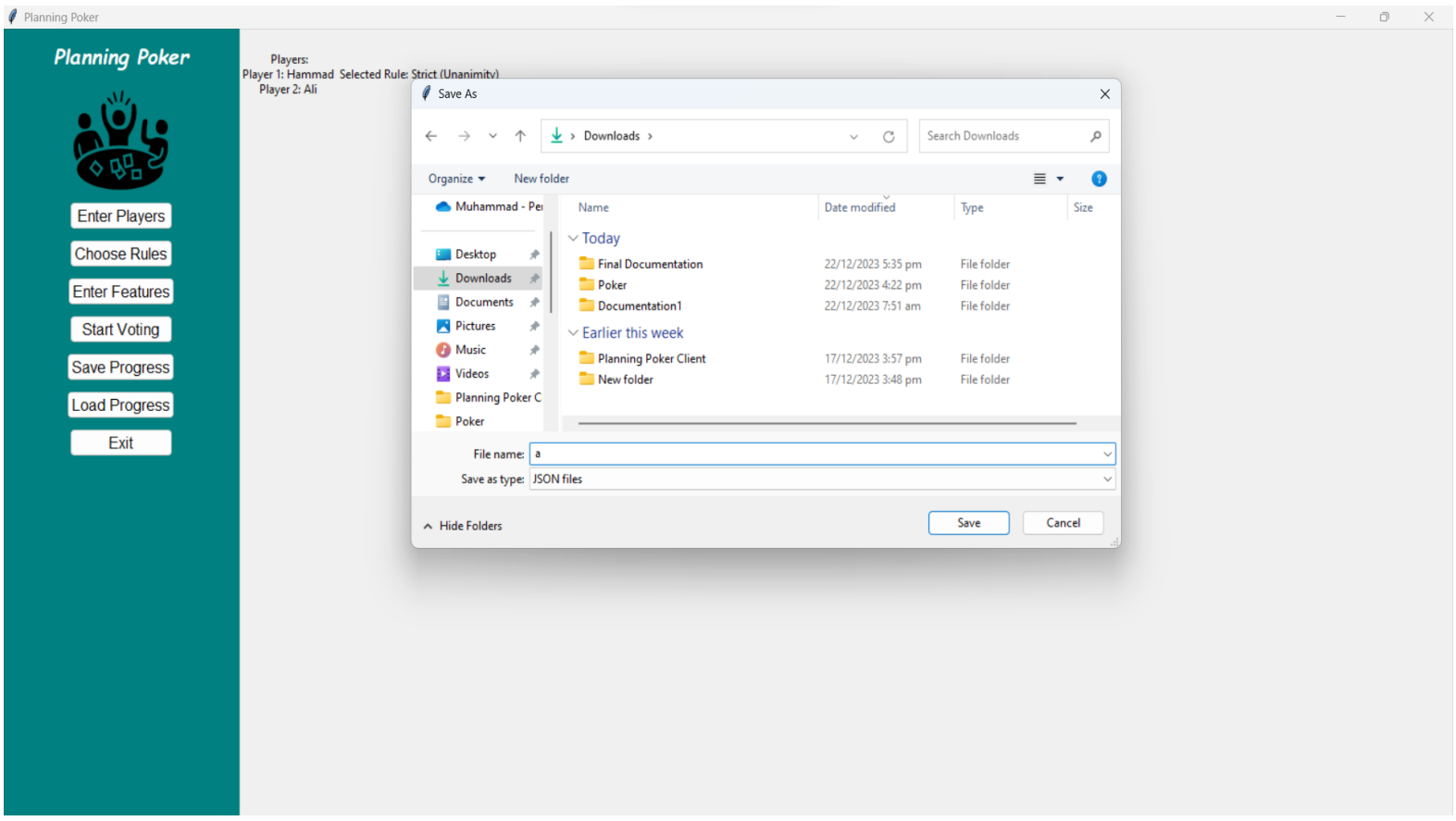

Figure 3.8 Save Progress

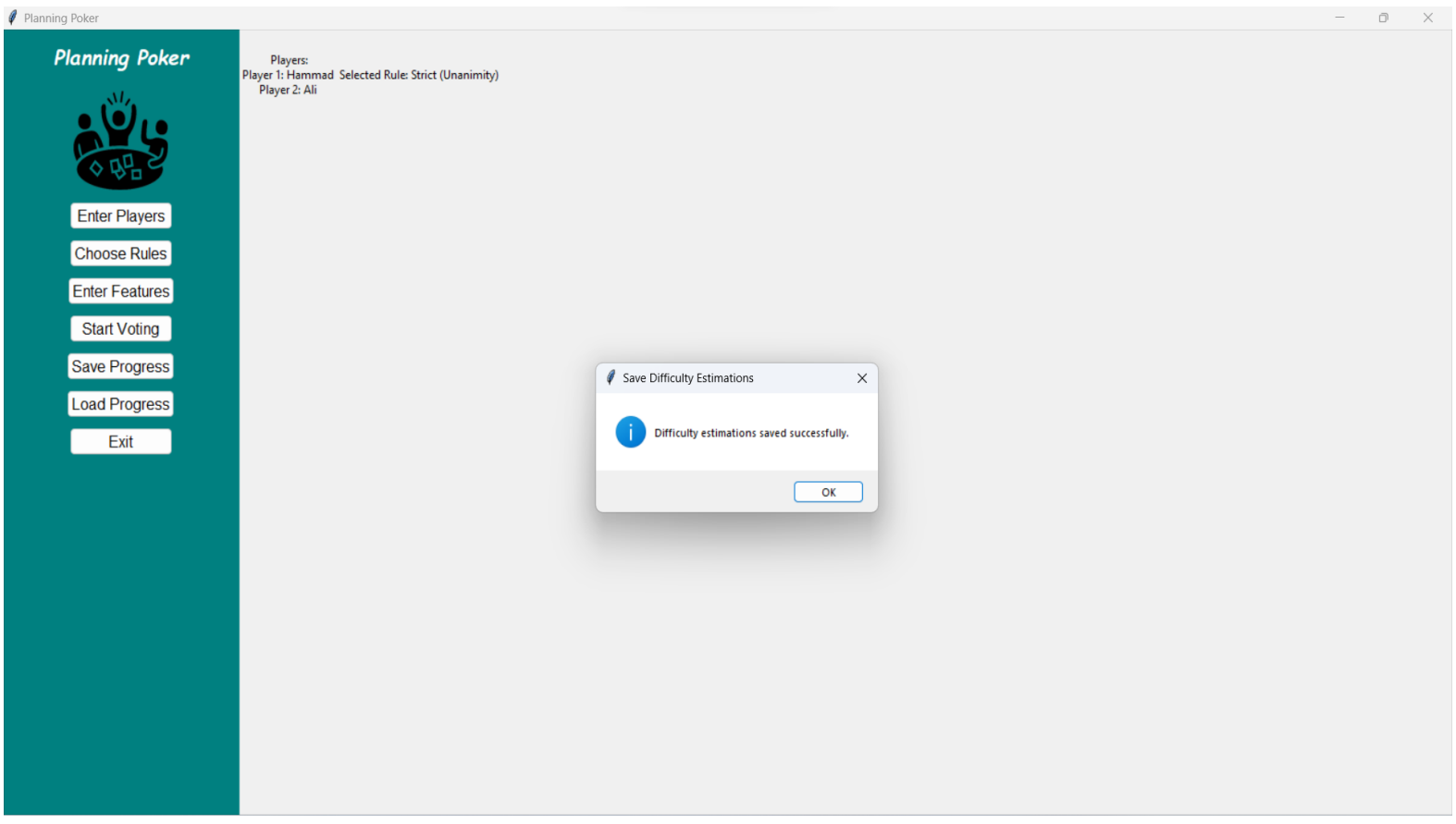

Figure 3.9 Confirmation of Save Progress

3.1.6.2 Loading Progress

This includes saving the progress of planning Poker game allows users to resume a previous session, providing continuity and a sense of persistence.

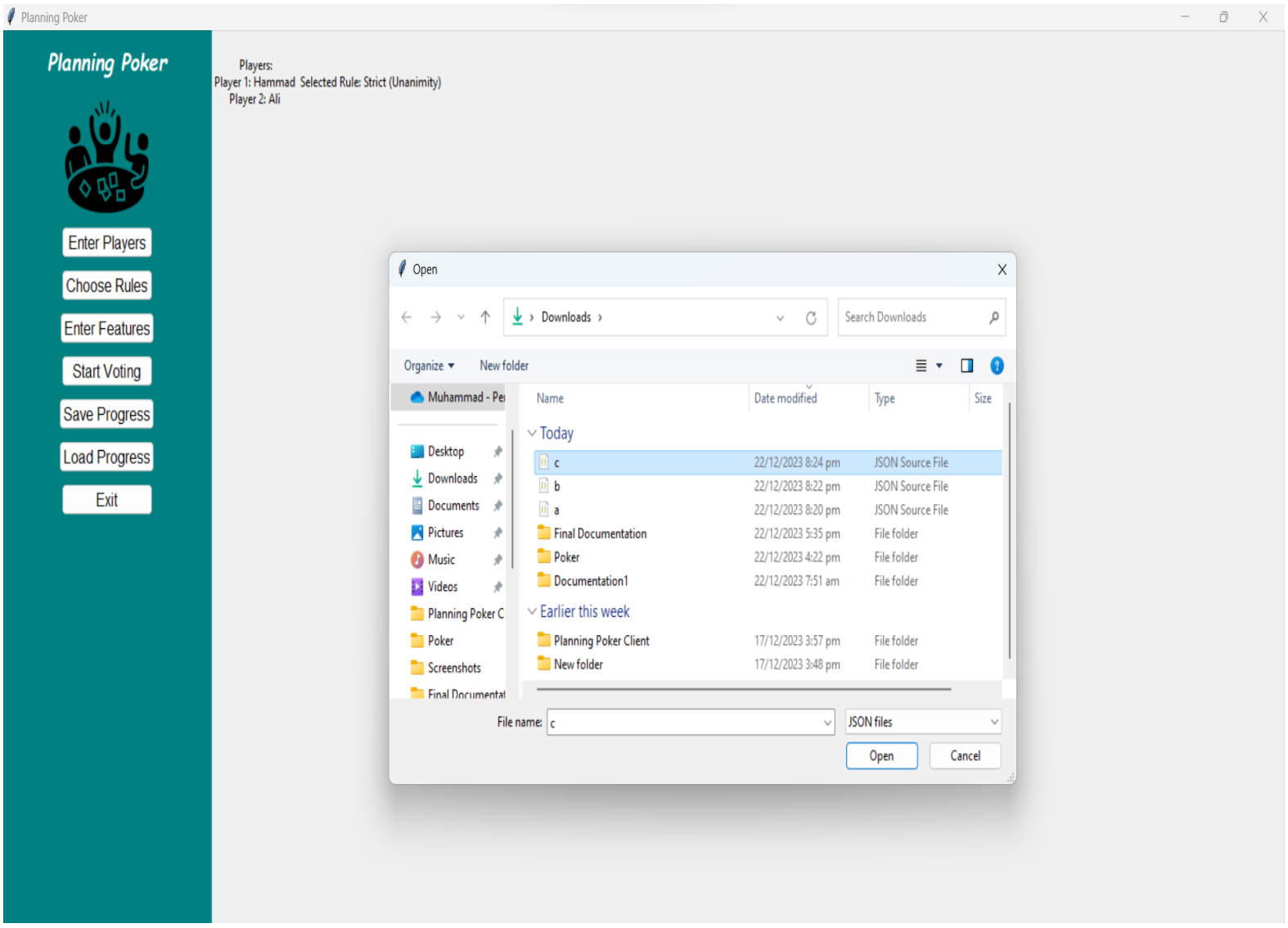

Figure 3.10 Load Progress

3.2 Usage Design Pattern

I have used design patterns in developing planning poker games.

3.2.1 MVC Design Pattern

The selection of the MVC (Model-View-Controller) design pattern is driven by several factors, each contributing to its importance in software development:

- **Separation of Concerns:** The division of code into distinct components, namely the Model, View, and Controller, facilitates a clear organization of responsibilities. This separation allows for a cleaner codebase where each component addresses a specific aspect of the application, leading to improved code readability and maintainability.
- **Maintainability:** The modular structure inherent in MVC promotes easier code maintenance. As each component focuses on a specific functionality, updates or modifications can be made to one part without significantly impacting the others. This characteristic simplifies the maintenance process and contributes to the longevity of the software.
- **Reusability:** The ability to reuse components across the application or in different projects is a crucial aspect of MVC. By isolating functionalities into separate modules, developers can leverage existing code, reducing redundancy and saving development time. This reusability enhances the efficiency and consistency of the software development process.
- **Testability:** MVC's support for independent testing of each component is instrumental in achieving high code quality. Developers can write unit tests for the Model, View, and Controller separately, ensuring that each part functions correctly. This promotes a robust testing strategy, resulting in a more reliable and stable software product.
- **Scalability:** The scalability afforded by MVC is vital for applications that may undergo growth or modification over time. With the ability to extend or modify individual components independently, developers can adapt the application to changing requirements without having to overhaul the entire system. This flexibility is crucial for the long-term success of the software.

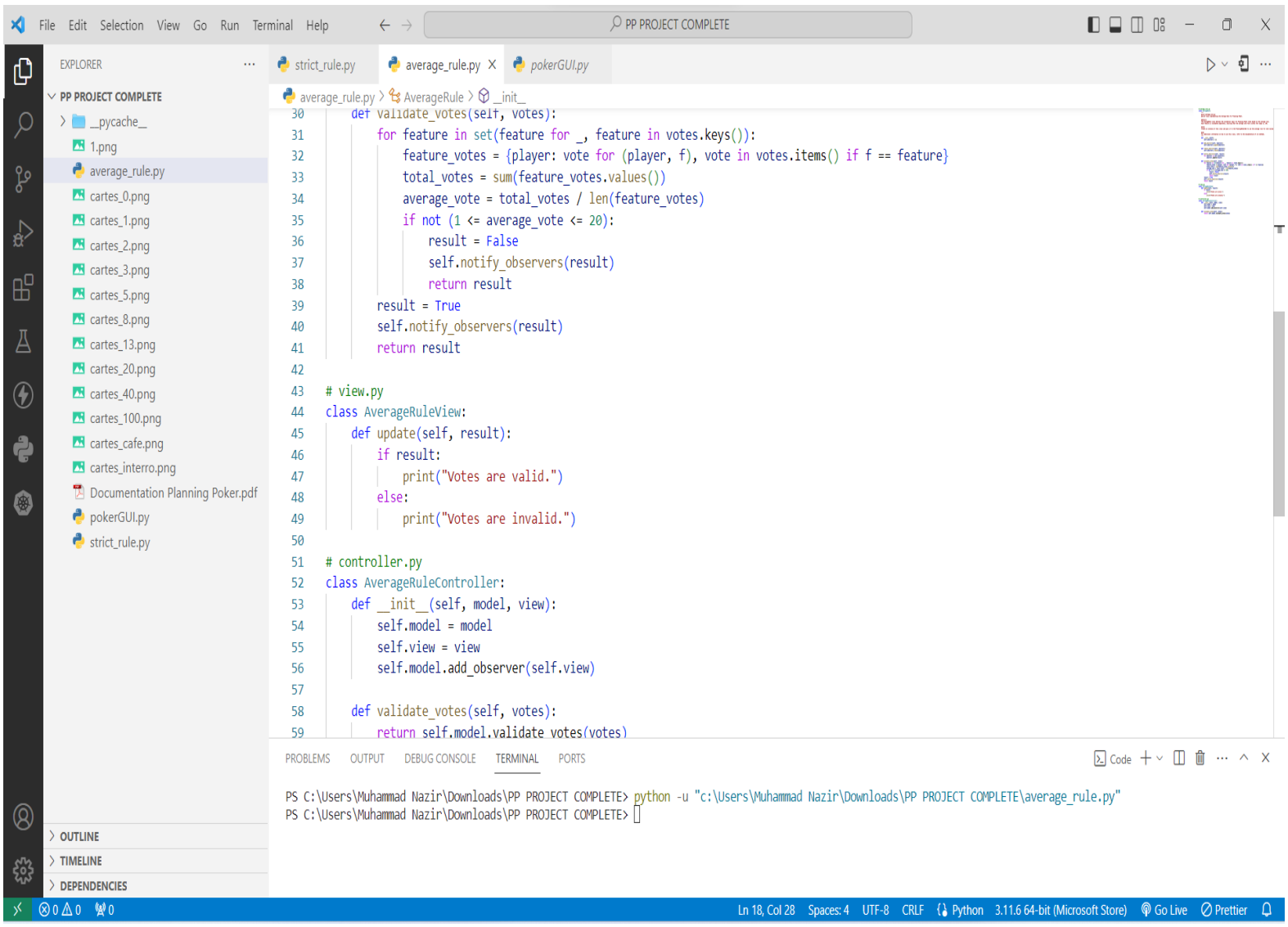

Figure 3.11 MVC Snap

3.2.2 Singleton Design Pattern

The adoption of the Singleton design pattern is grounded in several considerations, each contributing to its significance in software development:

- **Single Point of Control:** The Singleton pattern ensures that only one instance of a class exists, offering centralized control over a specific resource or service. This centralized control simplifies management and coordination within the system.
- **Global Access:** By providing a global point of access to a single instance, the Singleton pattern facilitates coordination and communication across different parts of the application. This global accessibility is particularly valuable when a consistent resource or service is needed throughout the system.
- **Resource Sharing:** The Singleton pattern efficiently manages and shares resources across the system. This can be advantageous in scenarios where efficient resource utilization is critical, preventing unnecessary duplication and enhancing overall system performance.
- **Lazy Initialization:** Singleton allows for lazy initialization, meaning that the object is created only when it is first requested. This saves resources and is beneficial when the instantiation of the object is resource-intensive or time-consuming.
- **Thread Safety:** The Singleton pattern can be designed to be thread-safe, preventing concurrency issues in multi-threaded environments. This ensures that multiple threads can safely access and use the singleton instance without conflicts.
- **Memory Management:** Singleton provides efficient memory usage by having only one instance throughout the application's lifecycle. This is particularly useful in scenarios with limited resources or when dealing with large objects.
- **Consistent State:** The Singleton pattern maintains a uniform state throughout the application. Changes to the state of the singleton are reflected consistently across all parts of the system, ensuring coherence and reliability.
- **Configuration Management:** Singleton is useful for managing configuration settings, providing a single source of truth for configuration parameters. This centralized approach simplifies configuration management and ensures consistency in the application's behavior.

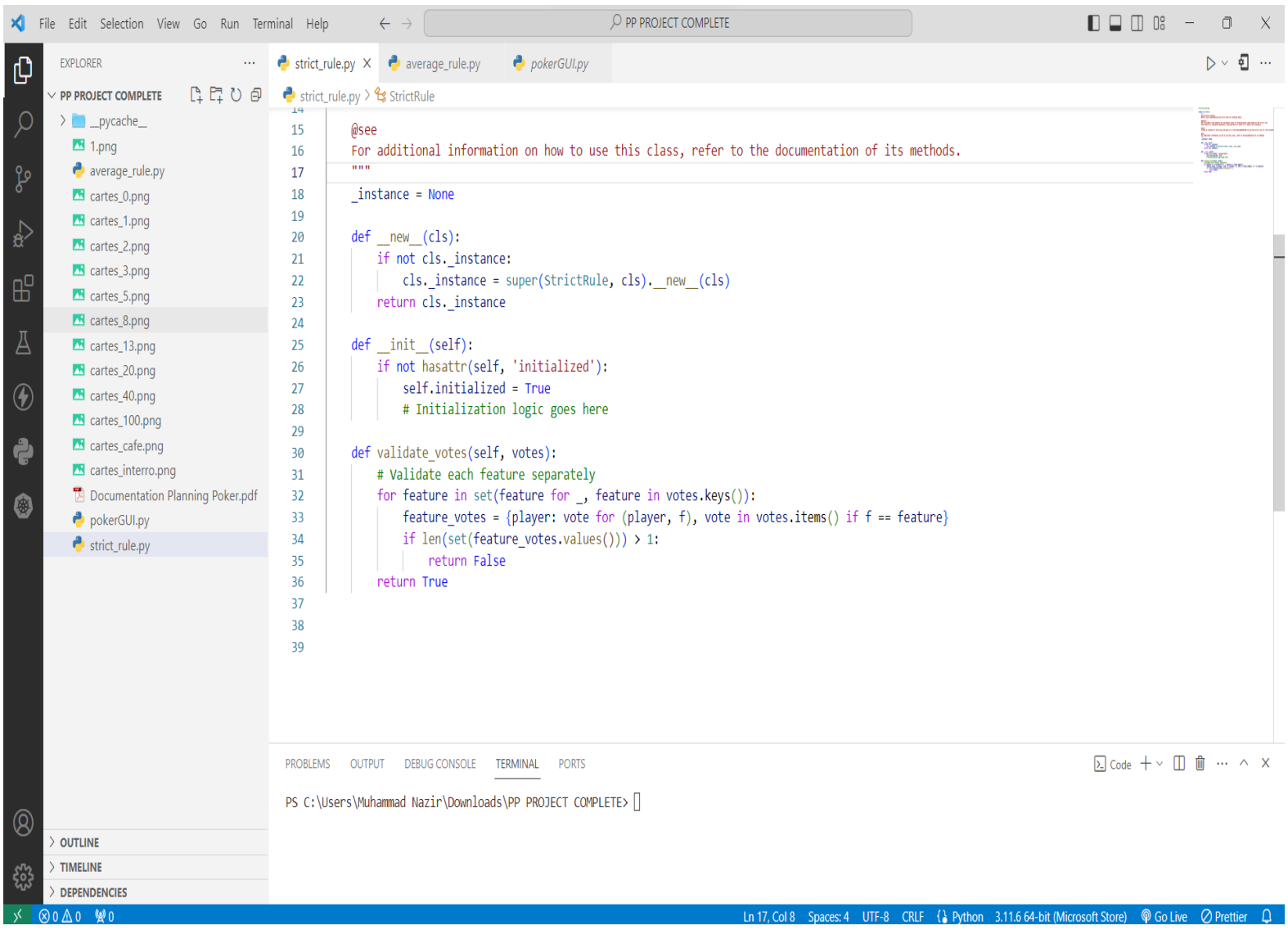

Figure 3.12 Singleton Snap

Chapter 4

Code Explanation

4.1 average_rule.AverageRule Class Reference

Public Member Functions

- **__init__** (self)
- **validate_votes** (self, votes)

4.1.1 Detailed Description

```
@file average_rule.py
@brief Class implementing the Average Rule for Planning Poker.

@details
The AverageRule class defines the validation logic for Planning Poker votes based on the average rule.
Each feature is validated separately, ensuring that the average vote falls within the range [1, 20].

@note
Create an instance of this class and pass it to the PlanningPokerGUI to use the average rule for vote validation.

@see
For additional information on how to use this class, refer to the documentation of its methods.
```

The documentation for this class was generated from the following file:

- average_rule.py

4.2 pokerGUI.PlanningPokerGUI Class Reference

Public Member Functions

- __init__ (self)
- create_menu (self)
- set_initial_size (self)
- enter_players_count (self, window)
- process_players_input (self, window, num_players)
- process_player_names (self, players_input, num_players, window)
- update_players_label (self)
- choose_rules (self)
- process_rule_selection (self, window, selected_rule)
- enter_features (self)
- process_features_input (self, features_input, window)
- set_vote_result (self, player, result_entry, feature, label_text)
- submit_votes (self)
- start_voting (self)
- submit_vote (self, vote_text, label_text)
- evaluate_votes (self)
- save_difficulty_estimations (self)
- update_vote_text (self, value)
- set_button_value (self, button_value, vote_text)
- close_voting_screen (self)

Public Attributes

- **root**
- **set_initial_size**
- **players**
- **features**
- **votes**
- **rules**
- **voting_window**
- **current_player_index**
- **current_feature_index**
- **left_frame**
- **main_frame**
- **selected_rule_label**
- **top_labels**
- **players_label**
- **choose_rules**
- **enter_features**
- **start_voting**
- **save_progress**
- **load_progress**
- **vote_texts**

4.2.1 Detailed Description

```
@brief Class for the Planning Poker GUI application.

@details
This class represents the graphical user interface (GUI) for a Planning Poker application.
It includes attributes to manage the main window, players, features, votes, rules, and the voting process.

@note
To use this class, create an instance and call the ‘run‘ method to start the Tkinter main loop.

@warning
Make sure to set the necessary attributes such as players, features, and rules before starting the voting process.

@code
planning_poker_gui = PlanningPokerGUI()
planning_poker_gui.run()
@endcode

@see
For additional information on how to use this class, refer to the documentation of its methods.

@par Attributes:
- root: The main Tkinter window.
- players: List of player names.
- features: List of features (backlog).
- votes: Dictionary to store votes.
- rules: Rules object for voting validation.
- voting_window: Tkinter window for the voting process.
- current_player_index: Index to keep track of the current player.
- current_feature_index: Index to keep track of the current feature.
- left_frame: Tkinter Frame for the left side of the main window.
- main_frame: Tkinter Frame for the main content.
- selected_rule_label: Tkinter Label to display the selected rules.
- top_labels: List to store top labels for display.
```

4.2.2 Constructor & Destructor Documentation

4.2.2.1 __init__()

```
pokerGUI.PlanningPokerGUI.__init__ (
              self )

@brief Constructor for the PlanningPokerGUI class.

Initializes the main Tkinter window and sets up the GUI elements.
```

4.2.3 Member Function Documentation

4.2.3.1 choose_rules()

```
pokerGUI.PlanningPokerGUI.choose_rules (
              self )

@brief Create a window for selecting planning poker rules.

This method creates a window for selecting planning poker rules. The window includes a label with instructions,
a combobox for choosing from a predefined list of rules, and a confirmation button.
The selected rule is then processed using the ’process_rule_selection’ method.

@return void
```

4.2.3.2 close_voting_screen()

```
pokerGUI.PlanningPokerGUI.close_voting_screen (
              self )
```

@brief Close the voting screen window.

This method checks if the voting window is open and closes it if it exists.
It sets the voting window attribute to None after closing.

@return void

4.2.3.3 create_menu()

```
pokerGUI.PlanningPokerGUI.create_menu (
              self )
```

@brief Create the menu section on the left side of the main window.

Creates labels, buttons, and other GUI elements for user interaction.

4.2.3.4 enter_features()

```
pokerGUI.PlanningPokerGUI.enter_features (
              self )
```

@brief Create a window for entering features in JSON format.

This method creates a window for entering features (backlog) in JSON format. The window includes a label with instructions,
a text entry field for typing the features, and a confirmation button.
The entered features are then processed using the 'process_features_input' method.

@return void

4.2.3.5 enter_players_count()

```
pokerGUI.PlanningPokerGUI.enter_players_count (
              self,
              window )
```

@brief Prompt the user to enter the number of players.

This method creates a new Toplevel window for entering the number of players.
It includes a label, an entry field, and a confirmation button.
The confirmation button triggers the processing of player input.

@param window The Tkinter window for player count input.

@return void

4.2.3.6 evaluate_votes()

```
pokerGUI.PlanningPokerGUI.evaluate_votes (
              self )
```

@brief Evaluate and process the collected votes.

This method prints the collected votes to the console and uses the specified rules to validate the votes.
If the votes are approved, it displays a message with the voting result, and you can customize the logic for further actions.
If the votes are not approved, it shows a warning message, clears the votes, and restarts the voting process.

@return void

4.2.3.7 process_features_input()

```
pokerGUI.PlanningPokerGUI.process_features_input (
              self,
              features_input,
              window )
```

```
@brief Process the entered features in JSON format and update the features attribute.

This method takes the entered features in JSON format and the Tkinter window for feature entry as input.
It destroys the feature entry window and attempts to parse the entered JSON features.
If successful, it updates the 'features' attribute; otherwise, it displays an error message.

@param features_input Entered features in JSON format.
@param window Tkinter window for feature entry.

@return void
```

4.2.3.8 process_player_names()

```
pokerGUI.PlanningPokerGUI.process_player_names (
              self,
              players_input,
              num_players,
              window )
```

```
@brief Process the entered player names and update the players list.

This function takes the entered player names, the expected number of players, and the Tkinter window for player names input.
It destroys the input window and performs validation on the entered player names.
If no players are entered, a warning message is displayed, and the function returns.
If the number of entered players does not match the expected number, a warning message is shown, and the function returns.
Otherwise, the players list is updated, and the method 'update_players_label' is called to reflect the changes.

@param players_input Entered player names separated by commas.
@param num_players Number of players expected.
@param window Tkinter window for player names input.

@return void
```

4.2.3.9 process_players_input()

```
pokerGUI.PlanningPokerGUI.process_players_input (
              self,
              window,
              num_players )
```

```
@brief Process user input for the number of players and set up player name entry.

This function takes the graphical user interface window and the number of players as input.
It attempts to convert the provided number of players to an integer and validates if it is a positive integer.
If the input is invalid, warning messages are displayed, and the function returns.
If the input is valid, the function dynamically creates a label and an entry widget for player names.
Additionally, a "Confirm" button is created with a callback function to process the entered player names.
@param window The graphical user interface window in which the input will be processed.
@param num_players The user-provided input indicating the desired number of players.

@return void
```

4.2.3.10 process_rule_selection()

```
pokerGUI.PlanningPokerGUI.process_rule_selection (
              self,
              window,
              selected_rule )
```

```
@brief Process the selected planning poker rule and update the rules attribute.

This method takes the Tkinter window for rule selection and the selected planning poker rule as input.
It destroys the rule selection window and updates the 'rules' attribute based on the selected rule.
The method then updates the labels displaying the selected rule in the GUI.

@param window Tkinter window for rule selection.
@param selected_rule Selected planning poker rule.

@return void
```

4.2.3.11 save_difficulty_estimations()

```
pokerGUI.PlanningPokerGUI.save_difficulty_estimations (
              self )
```

@brief Save the difficulty estimations based on the collected votes to a JSON file.

This method prompts the user to choose a file path for saving the difficulty estimations in JSON format.
It calculates the average vote for each feature and saves the difficulty estimations to the specified file.
After saving, it displays an information message indicating the successful save.

@return void

4.2.3.12 set_button_value()

```
pokerGUI.PlanningPokerGUI.set_button_value (
              self,
              button_value,
              vote_text )
```

@brief Set the corresponding button value in the text box.

This method sets the corresponding button value in the text entry field for the current player's vote.
The value is appended to the existing text.

@param button_value The value associated with the clicked button.
@param vote_text The text entry field for the current player's vote.

@return void

4.2.3.13 set_initial_size()

```
pokerGUI.PlanningPokerGUI.set_initial_size (
              self )
```

@brief Set the initial size of the main window.

Called after a delay to adjust the size of the main window.

4.2.3.14 set_vote_result()

```
pokerGUI.PlanningPokerGUI.set_vote_result (
              self,
              player,
              result_entry,
              feature,
              label_text )
```

@brief Set the vote result for a player on a specific feature.

This method takes the player, result entry, feature, and label text as input.
It collects and stores the vote result for the given player and feature.
After each vote, it checks if all votes are collected before evaluating.
It then updates the indices for the next set of widgets and clears the content of the text box.
If all features or votes are collected, it resets the corresponding indices and updates the label text accordingly.
If all votes are collected, it automatically submits the votes.

@param player The player casting the vote.
@param result_entry The text entry field where the vote result is entered.
@param feature The feature for which the vote is being cast.
@param label_text The label text to be updated for the next set of widgets.

@return void

4.2.3.15 start_voting()

```
pokerGUI.PlanningPokerGUI.start_voting (
              self )
```

@brief Start the voting process by creating a window for voting.

This method initiates the voting process, ensuring that players, features, and rules are entered.
It resets the votes dictionary and indices, creates a new Toplevel window for voting, or updates the existing one.
The window includes labels for current voting information, text widget for voting, and buttons for submitting votes.
PNG images are used on buttons for different vote values.

@return void

4.2.3.16 submit_vote()

```
pokerGUI.PlanningPokerGUI.submit_vote (
              self,
              vote_text,
              label_text )
```

@brief Submit a vote and handle the voting process.

This method takes a text entry field for voting (‘vote_text‘) and the label text for updating the next set of widgets.
It inserts the selected vote into the corresponding player’s text box and checks if all votes are collected before evaluating.
If all votes are collected, it triggers the evaluation process. Otherwise, it moves to the next player and updates the label.
If all features have been voted on, it shows the result; otherwise, it updates the label for the next set of widgets.

@param vote_text The text entry field containing the selected vote.
@param label_text The label text for updating the next set of widgets.

@return void

4.2.3.17 submit_votes()

```
pokerGUI.PlanningPokerGUI.submit_votes (
              self )
```

@brief Submit and handle the collected votes.

This method implements the logic to handle the submitted votes.
In this example, it prints the collected votes to the console.

@return void

4.2.3.18 update_players_label()

```
pokerGUI.PlanningPokerGUI.update_players_label (
              self )
```

@brief Update the label displaying the list of players.

This method updates the label that displays the list of players.
It generates a formatted text containing the player names and their corresponding indices, and updates the GUI accordingly.

@return void

4.2.3.19 update_vote_text()

```
pokerGUI.PlanningPokerGUI.update_vote_text (
              self,
              value )
```

@brief Update the vote text with the selected value.

This method takes a value and updates the text entry field for the current player’s vote.
The value is inserted at the end of the current text.

@param value The selected value to be added to the vote text.

@return void

The documentation for this class was generated from the following file:

- pokerGUI.py

4.3 strict_rule.StrictRule Class Reference

Public Member Functions

- **__init__** (self)
- **validate_votes** (self, votes)

4.3.1 Detailed Description

```
@file strict_rule.py
@brief Class implementing the Strict Rule for Planning Poker.

@details
The StrictRule class defines the validation logic for Planning Poker votes based on the strict rule.
Each feature is validated separately, ensuring that all votes for a feature are unanimous.

@note
Create an instance of this class and pass it to the PlanningPokerGUI to use the strict rule for vote validation.

@see
For additional information on how to use this class, refer to the documentation of its methods.
```

The documentation for this class was generated from the following file:

- strict_rule.py

Index