Intelligent Briscola

(v. 0.1.0)

Andrea Zammarchi
andrea.zammarchi3@studio.unibo.it

June 19, 2024

Intelligent Briscola aims to develop the Italian classic card game Briscola where users can play against a sophisticated bot designed as an intelligent agent. The objective is to create a dynamic and challenging gameplay experience where the bot simulates human-like strategies in the card game. The project will involve the implementation of different levels of bot intelligence. Each level will employ varying degrees of strategic decision-making processes, allowing the bot to adapt its gameplay according to the intelligence level selected by the user. The intelligent agent will utilize advanced algorithms to analyze the state of the game and make optimal decisions, providing a realistic and competitive gaming experience. The matches will exclusively consist of two players, in a 1vs1 mode (one human and one bot). The cards used to play Briscola are 40, grouped into 4 suits: clubs, coins, cups and swords. The type of cards will be the "Romagnola" variety. The total points available in the deck are 120; to declare a winner, it is necessary to reach a minimum of 61 points, while if 60 points are reached, a tie occurs.

1 Goals/requirements

Goals

- Implement a multi-agent system that includes different levels of intelligence for a card-playing bot.
- Define and implement autonomous behaviors for the bots, allowing them to make strategic decisions based on the context of the game and the difficulty level.
- Develop the complete game logic for Briscola, ensuring correct rules, scoring, and mechanics.
- Create an intuitive graphical user interface (GUI) for real-time game interaction and monitoring.

- Conduct thorough testing to ensure the correctness and effectiveness of the game logic.
- Focus on creating a user experience that is engaging and pleasant.

Expected outcomes

- A fully functional simulation of the Briscola card game where human players can play against an intelligent bot.
- Bots exhibit realistic behaviors, adapting their strategies based on the selected difficulty level.
- A visually appealing and user-friendly GUI that allows effective interaction and monitoring of the game.
- An engaging and challenging gaming experience that encourages repeated play and user satisfaction.

1.1 Scenarios

Use case scenarios

1. **Launching the application**:
   - **User Action**: Launch the game application.
   - **System Response**: The main menu appears, prompting the user to enter their name, select bot difficulty (Stupid, Normal, Intelligent), and choose the starting player option (Player, Bot, or Random by default).
   - **Why**: Users start here to set up their game preferences, ensuring a personalized experience.
2. **Configuring game settings**:
   - **User Action**: Enter the name, select the difficulty of the bot and choose the starting player.
   - **System Response**: The system records these inputs and prepares the game accordingly.
   - **Why**: Users configure settings to tailor the game to their preferences and desired challenge level.
3. **Starting the match**:
   - **User Action**: Press the "Start" button.
   - **System Response**: The system transitions to the match view, setting up the game board and dealing cards.
   - **Why**: Users initiate the game to begin playing Briscola.

4. **Playing the game**:
    - **User Action**: Play a card when it is their turn.
    - **System Response**: The system processes the move, updates the state of the game, and prompts the bot to play its turn.
    - **Why**: Users interact to progress through the game, applying strategies to win.
5. **Quitting the match**:
    - **User Action**: Press the "Quit" button to exit the match and return to the main menu.
    - **System Response**: The system navigates back to the main menu.
    - **Why**: Users might want to end the current match early and return to the menu for other options.
6. **Viewing the scoreboard**:
    - **User Action**: Press the "Scoreboard" button in the main menu.
    - **System Response**: The system displays a scoreboard showing all previous matches with player names, bot levels, and scores.
    - **Why**: Users can review past performances and compare results across different games.

Use case diagrams Figure 1 shows the Use Case Diagram for the menu view where the player can choose it's name, the bot's level of intelligence and who will start first.

Figure 2 shows the Use Case Diagram for the match view where the player can only play a card if it's his turn or can quit the game.

1.2 Self-assessment policy

Assessing the Quality of the produced software

1. **Code Quality**:
    - **Code reviews**: Regular reviews should be conducted to ensure the adherence to coding standards, readability, and maintainability.
    - **Static code analysis**: Utilize Kotlin-specific tools and plugins for static code analysis to detect bugs and security vulnerabilities.
    - **Unit testing**: Comprehensive unit tests should be written for all critical components, particularly focusing on the utils and model packages using the Kotlin test. Aim for high code coverage to ensure correctness and reliability.
2. **Performance Testing**:
    - **Load Testing**: Simulate multiple simultaneous games to evaluate the system's performance under stress and ensure it can handle peak loads.

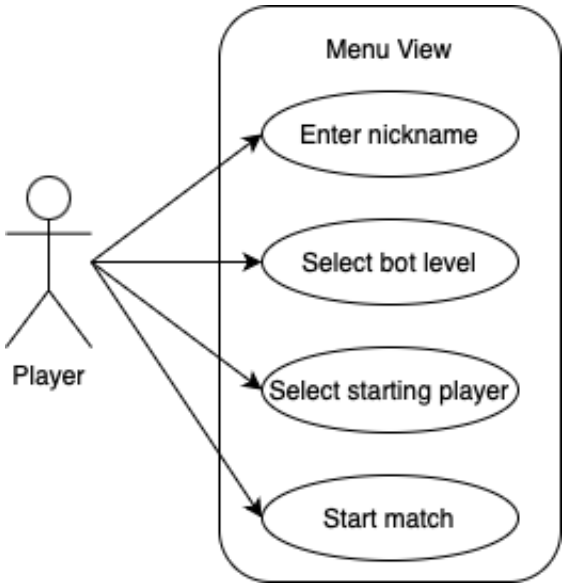

Figure 1: Menu View use case diagram

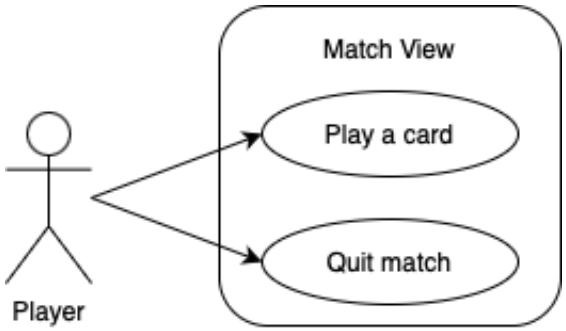

Figure 2: Match View use case diagram

- **Response Time**: Measure and optimize the system response time to ensure a smooth user experience.

3. **Usability Testing**:
   - **User Feedback**: Gather feedback from a diverse group of users to identify usability issues and areas for improvement.
   - **User Interface Evaluation**: Ensure that the interface is intuitive, easy to navigate, and provides a pleasant user experience.
4. **Documentation**:
   - **In-Code Documentation**: Code IS documentation. Use Kotlin's default documentation comments ("*/** comment */*") directly in the code to provide clear and concise explanations of functions, classes and modules. This ensures that the codebase is self-explanatory and easy to understand for other developers.

Assessing the Effectiveness of the project outcomes

1. **Functional Requirements**:
   - **Requirement Verification**: Regularly verify that all requirements are being met through systematic testing and validation.
2. **Autonomous Behavior Effectiveness**:
   - **Realism of Behavior**: Verify that bots exhibit realistic and context-aware behaviors, making strategic decisions based on the game state.
3. **Outcome Validation**:
   - **Match Outcome Consistency**: Ensure that match outcomes are consistent with the rules of Briscola and reflect accurate scoring.
   - **Scenario Testing**: Test the system with various game scenarios to ensure robustness and reliability in handling edge cases and unexpected situations.

2 Background

Briscola is a traditional Italian card game that requires strategic thinking and decision-making. The game involves two or more players (but for this project the focus is on 1v1) and uses a standard 40-card Italian deck. Understanding the rules and mechanics of Briscola is essential for appreciating the complexity and design of the system developed in this project.

Game Rules

- The players aim to score points by winning tricks that contain high-value cards. The total points available in the deck are 120 so, in order to win, a player must gain at least 61 points. The match can end in a draw if both players score 60 points.
- Each card has a suit:
  - *Clubs*
  - *Coins*
  - *Cups*
  - *Swords*

  And a rank, with a corresponding value:
  - Rank 1 (Ace) → Value 11
  - Rank 2 → Value 0
  - Rank 3 → Value 10
  - Rank 4 → Value 0
  - Rank 5 → Value 0
  - Rank 6 → Value 0
  - Rank 7 → Value 0
  - Rank 8 (Jack) → Value 2
  - Rank 9 (Horse) → Value 3
  - Rank 10 (King) → Value 4
- At the beginning of each match, a briscola suit is established which will prevail over all other suits. In turn, each player plays a card. In order to win the trick, a player can:
  - play a card with the briscola suit if the other has not done so;
  - play a card with the briscola suit and a greater value if the other also played briscola;
  - play a card with same suit of the played card but with higher value;
  - play a card with same suit of the played card but with higher rank if value is the same.

  In all other cases, whoever played the card first wins the trick.
- The one who wins a trick draws first in the next turn.
- A match ends when the players play all their cards in hands and deck.

3 Requirements Analysis

Implicit Requirements

- The software must strictly adhere to the rules of Briscola, ensuring accurate implementation of card ranking, trick-taking mechanics, and scoring.
- Players should be able to easily understand and follow the game flow without ambiguity.
- The interface must allow users to enter their name, select the bot difficulty, and choose the starting player before starting a match.
- During the match, the user should be able to play cards when it's their turn and quit the match to return to the main menu.
- The software should provide clear and immediate feedback for user actions, ensuring a seamless gaming experience.
- The system should support three levels of bot intelligence (stupid, normal, intelligent) to cater to different user skill levels.
- Bots should exhibit appropriate strategic behaviors based on the selected difficulty level.

Non-functional Requirements

- The game should load quickly, with minimal latency during gameplay to ensure a smooth user experience.
- The graphical user interface should be intuitive and easy to navigate, with clear instructions and prompts.
- The design should be user-friendly, catering to both novice and experienced players.
- The system should be robust, handling unexpected inputs or actions gracefully without crashing.
- The codebase should be well-documented.
- The system should be modular, allowing for easy updates and enhancements without significant refactoring.
- The software should be compatible with various operating systems and devices, ensuring accessibility for a wide range of users.

Model/Paradigm/Technology

- **Kotlin**
  - **Motivation for choice**: Kotlin provides a concise, expressive syntax, and robust safety features, making it an ideal choice for developing maintainable and reliable applications.
  - **Usage**: Kotlin is used for implementing the core game logic.
- **Jason**
  - **Motivation for choice**: Jason facilitates the development of sophisticated multi-agent systems with clear organizational structures and interaction protocols.
  - **Usage**: It is used to manage the autonomous behavior of the bots, enabling them to make strategic decisions based on the game context and difficulty level.
- **Kotlin Test**
  - **Motivation for choice**: Kotlin Test offers a robust framework for writing and running unit tests, supporting various testing styles, and providing comprehensive test coverage.
  - **Usage**: Kotlin Test is utilized for testing the `utils` and `model` packages, ensuring the correctness and reliability of the game logic.
- **Kotlin Plugins for QA**
  - **Motivation for choice**: Detekt provides powerful static code analysis and style checking, helping maintain high code quality.
  - **Usage**: This plugin is integrated into the development workflow to continuously monitor and improve code quality.

4 Design

The system is built on a centralized architecture, managed through a central entity called the *Multi-Agent System* (**MAS**), which coordinates all activities of the agents. The system includes a user-friendly graphical interface (GUI) for monitoring and interacting with the game. The system comprises three levels of bot agents (stupid, normal, intelligent) responsible for simulating different levels of AI gameplay. This centralized design ensures efficient coordination and provides a seamless gaming experience.

To facilitate the interaction between the agents and the game environment, a class named `BriscolaEnvironment` was developed. This class encapsulates the game environment and provides various methods and properties essential for managing the game state and interactions.

4.1 Structure / Domain Entities

The project is organized into several packages, each serving a specific role in the overall architecture (see Figure 3).

Motivation for design choice This design was chosen for several reasons:

- **Separation of Concerns**: By dividing the project into distinct packages, each responsible for specific aspects of the application, the design promotes clean separation of concerns. This makes the codebase easier to understand, maintain, and extend.
- **Modularity**: The modular design allows for independent development and testing of different components. For instance, the game logic can be developed and tested separately from the GUI and agent behaviors.
- **Scalability**: The structure supports scalability, making it easier to add new features or components, such as additional bot intelligence levels or new game rules, without affecting other parts of the system.
- **Reusability**: Utility classes and functions are placed in a dedicated package, promoting reusability across different parts of the application.
- **Maintainability**: The clear organization and separation of logic facilitate easier maintenance and debugging, as issues can be isolated within specific packages.

Similarity to MVC Pattern The project's package structure bears similarity to the *Model-View-Controller* (**MVC**) pattern, which is a widely used design pattern in software development. In MVC:

- **Model**: The `Model` package corresponds to the "Model" in MVC, representing the data and business logic of the game, such as the rules of Briscola and the state of the game.
- **View**: The `View` package corresponds to the "View" in MVC, managing the graphical user interface and presenting the game state to the user.
- **Controller**: The Controller role is divided between the `asl` and the `env`, which handle the interaction logic, agent behaviors, and coordination between the model and the view.

Figures 4, 5, 6 and 7 represent the UML Class Diagram of the system's entities, in particular all the classes contained in `env`, `model`, `view` and `utils` package.

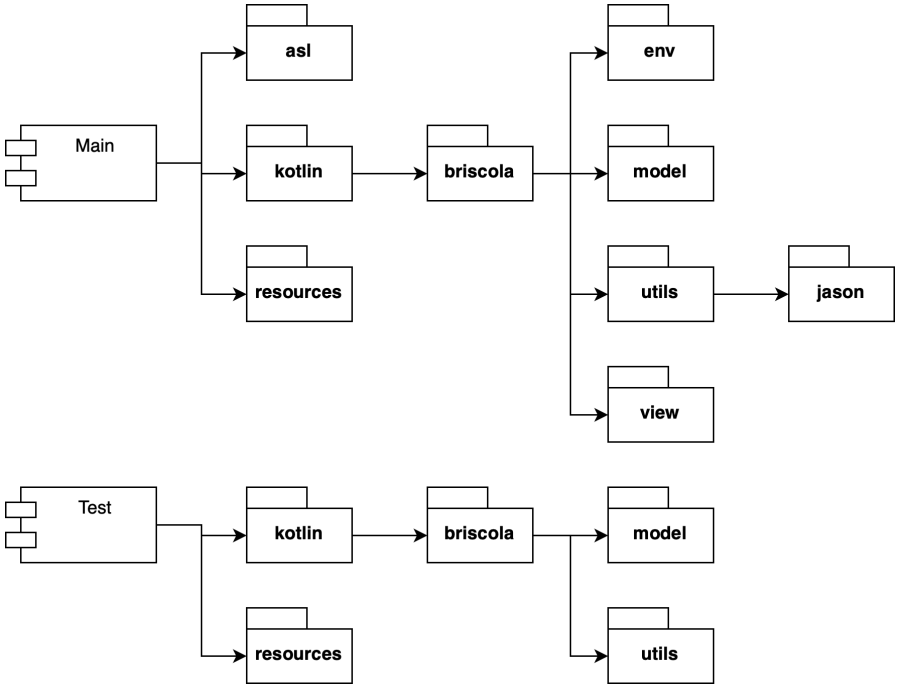

Figure 3: Package diagram

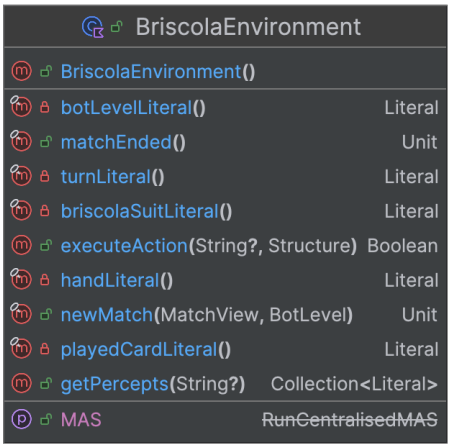

Figure 4: `Environment` package class diagram

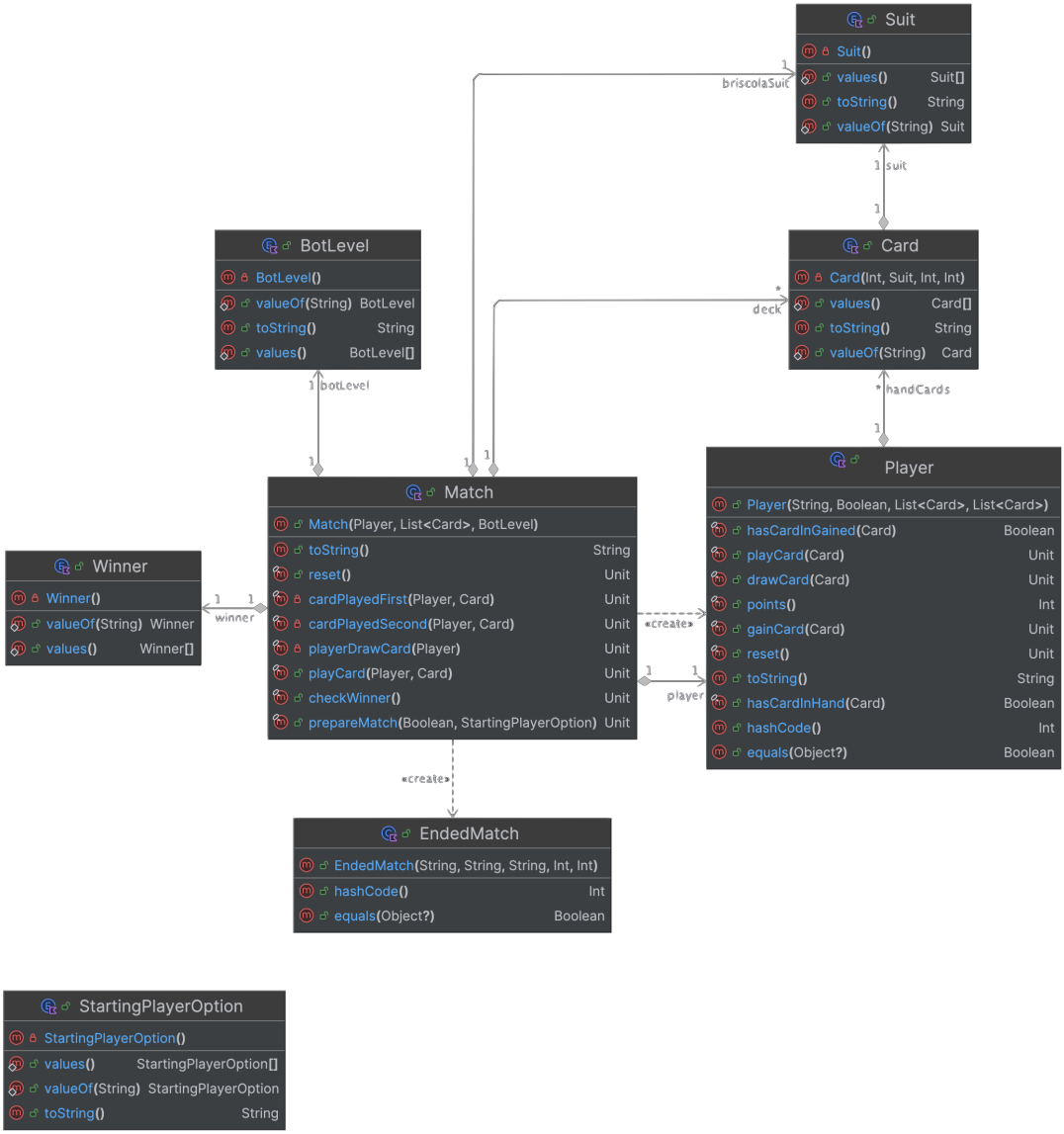

Figure 5: `Model` package class diagram

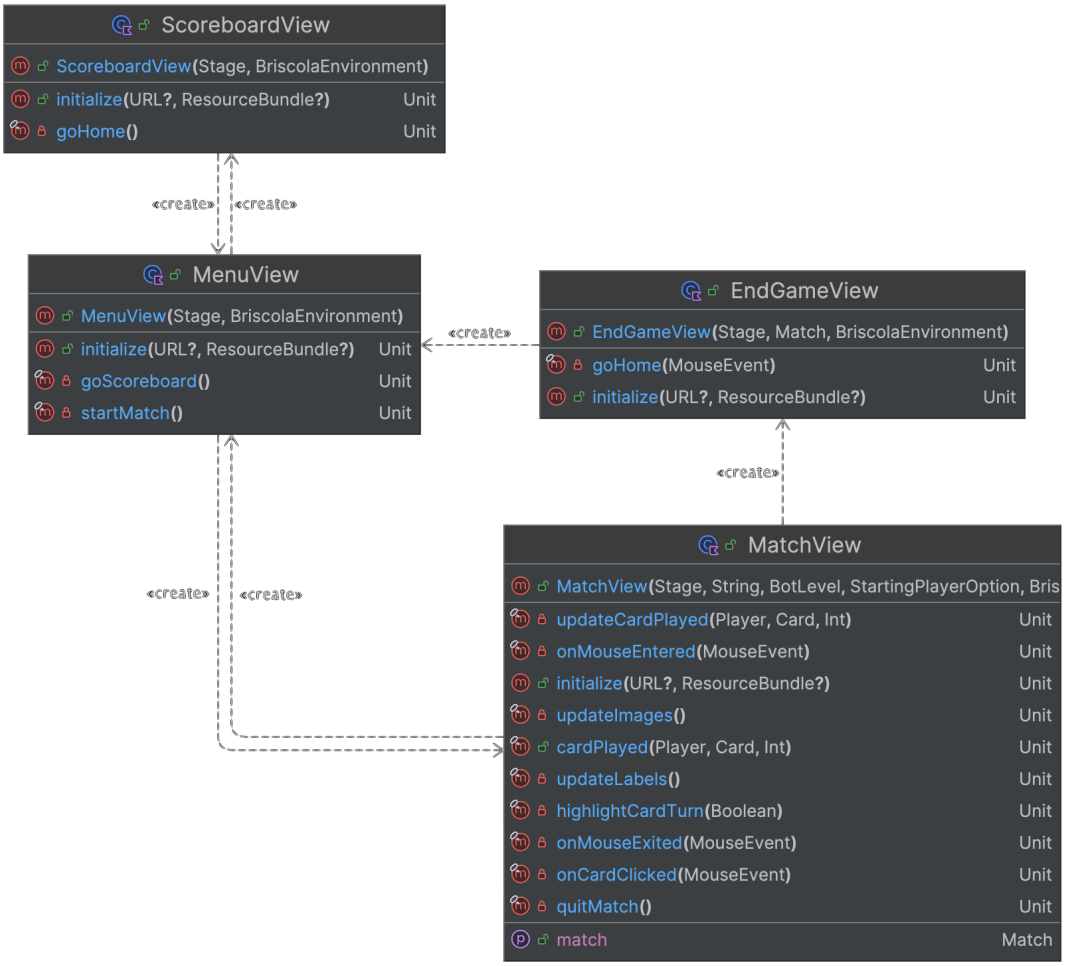

Figure 6: `View` package class diagram

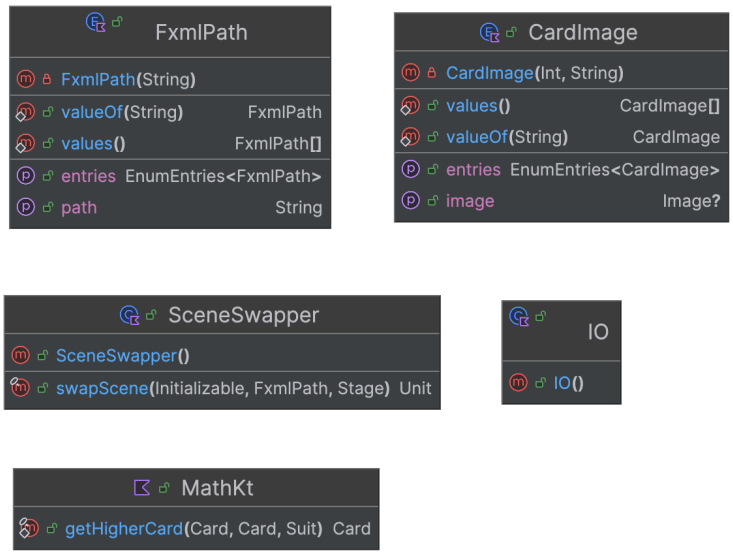

Figure 7: `Utils` package class diagram

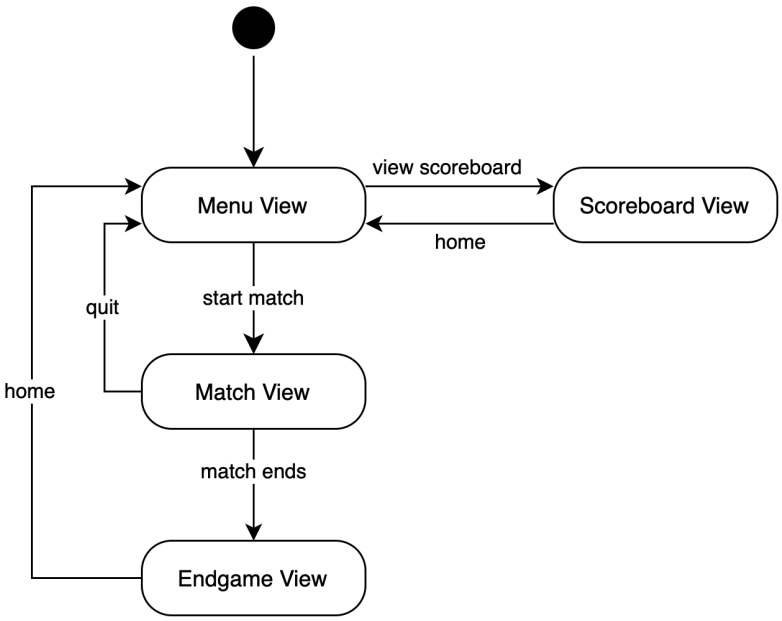

Figure 8: Views state diagram

4.2 Behaviour

Figure 8 shows the State Diagram from a user's perspective, from the moment he launches the application to the moment a match ends and is brought back to the home menu. The application can end gracefully at any time when the user decides to close it.

Figure 9 shows the State Diagram of the logic of a match. The logic of a game is the same as the official physical game of Briscola[1] and follows the same rules. In the event of sudden or deliberate disconnection, the server manages this event perfectly and acts accordingly based on the situation, that is, depending on whether the user was in a match or not.

Figure 10 illustrates the flow of activities of the *stupid bot* during a it's turn. The *stupid bot* employs a simplistic strategy without considering the value of the cards or the game state. As the first player, the bot simply plays a random card from its hand. Similarly, as the second player, the bot again plays a random card regardless of the card played by the opponent or the briscola suit. This approach ensures that the bot's actions are entirely unpredictable and do not involve any strategic decision-making.

Figure 11 illustrates the flow of activities of the *normal bot* during a it's turn. The *normal bot* employs a straightforward strategy to play the game. As the first player, it selects and plays the card with the lowest value, prioritizing cards with a suit different from the briscola to preserve its stronger cards. As the second player, the bot analyzes the card played by the opponent and attempts to play the lowest value card that can win the trick. If no such winning card is available, the bot defaults to playing the lowest card in its hand. This strategy ensures a basic level of strategic play, balancing between conserving valuable cards and trying to win tricks.

Figure 12 illustrates the flow of activities of the *intelligent bot* during a it's turn. The *intelligent bot* employs a sophisticated strategy to maximize its chances of winning. As the first player, it plays the card with the lowest value, prioritizing cards with a suit different from the briscola to conserve stronger cards. As the second player, the bot's decision-making process involves several steps:

- If the opponent plays a card with 0 value (and it is not briscola), the bot attempts to win the trick with a card of the same suit.
- If the opponent plays a card with value:
  - If the card is briscola, the bot tries to play a card with 0 value (not briscola), if possible, to avoid wasting valuable cards.
  - If the card is not briscola, the bot plays the lowest value card that can win the trick.
- If there are no higher cards available, the bot defaults to playing the lowest card in its hand.

This strategy allows the intelligent bot to effectively balance between conserving valuable cards, winning necessary tricks, and making calculated decisions based on the opponent's plays.

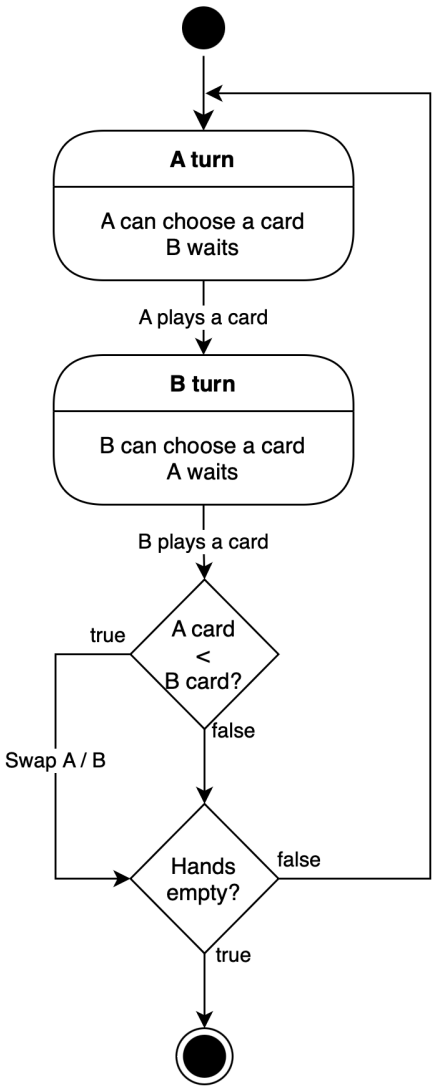

Figure 9: Match state diagram

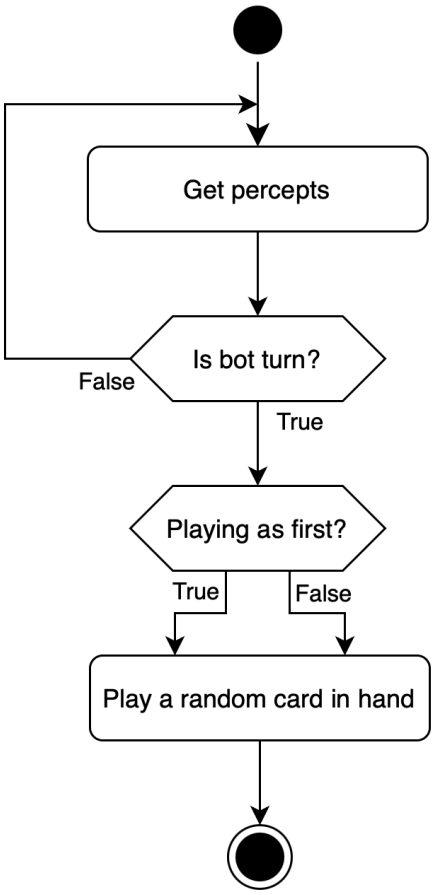

Figure 10: *stupid bot* turn activity diagram

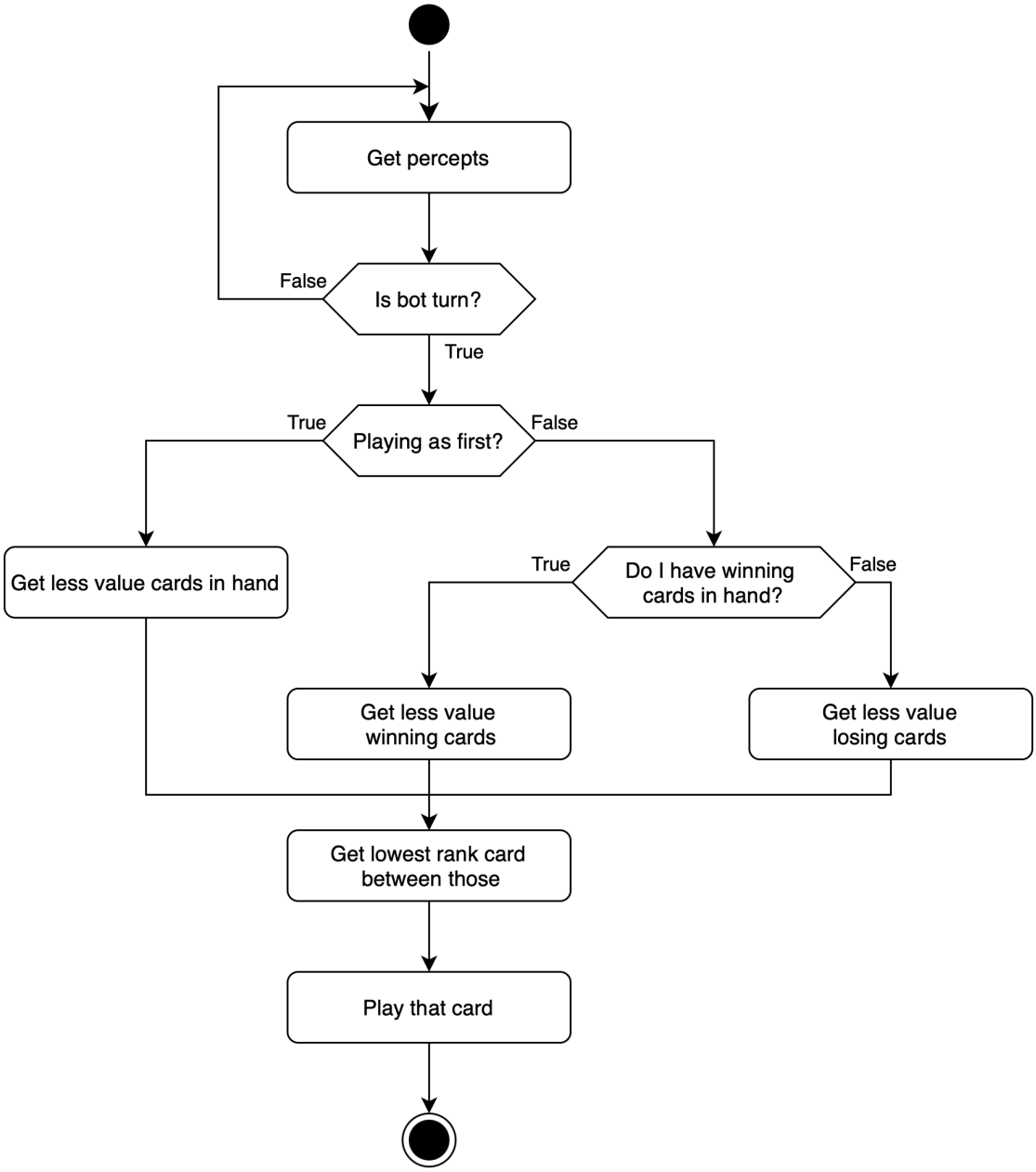

Figure 11: *normal bot* turn activity diagram

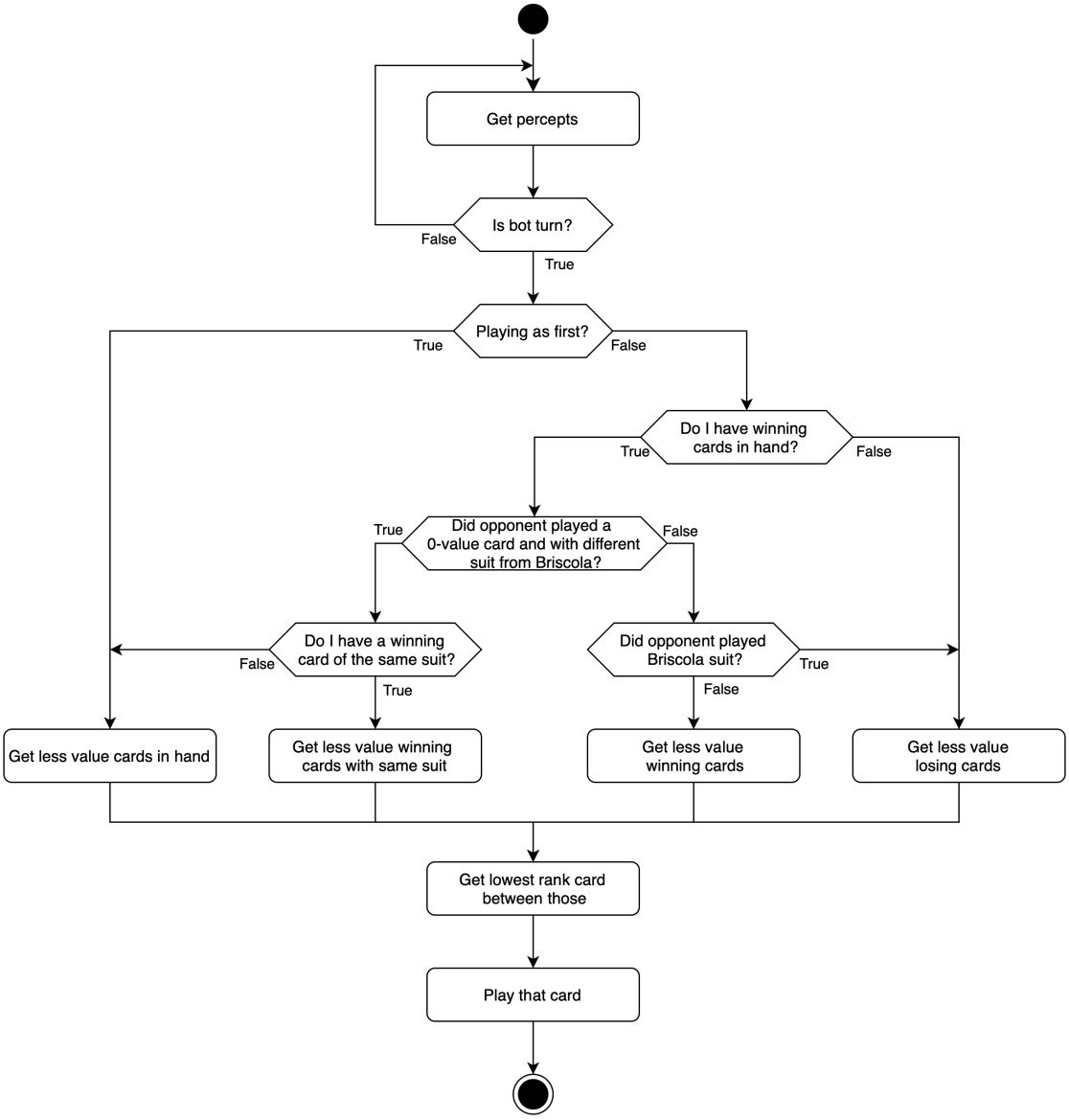

Figure 12: *intelligent bot* turn activity diagram

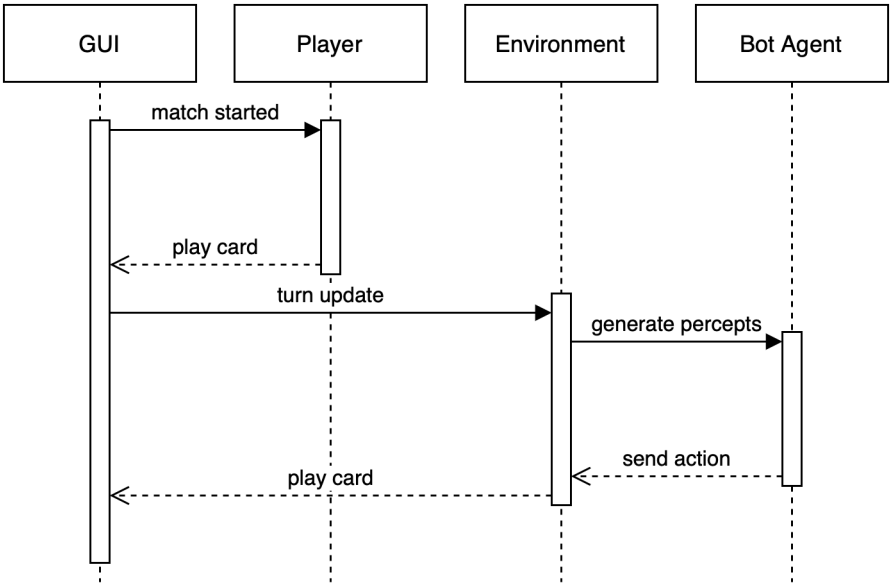

Figure 13: Match interactions sequence diagram

4.3 Interaction

In Intelligent Briscola system, the interaction between entities (player, bot, and the environment) is crucial for maintaining the flow of the game. The following UML Sequence Diagram (Figure 13) illustrates the interactions between the main entities during a match (to avoid repetition, the interactions are shown only for one turn).

Environment The `BriscolaEnvironment` plays a pivotal role in the interaction model. It is responsible for managing the state of the game and serving as the mediator between the agents and the game state. Here are the key percepts handled by the environment:

- `hand(List<Card>)`: This percept represents the current hand of cards held by the bot. It allows the bot to know which cards are available for play.
- `turn(BotTurn, NPlayedCards)`: This percept indicates whether it is the bot's turn and the number of cards that have been played in the current round. It helps the bot decide its actions based on whether it is the first or second player in the turn.
- `bot_level(Level)`: This percept indicates the intelligence level of the bot, allowing to "wake up" the correct bot agent to play against the player.
- `played_card(Card)`: This percept shows the card played by the opponent, which is crucial for the bot when deciding its move as the second player. The *stupid bot* ignores this percept.

- `briscola_suit(Suit)`: This percept indicates the suit of the briscola card, which can influence the bot's strategy in choosing which card to play. The *stupid bot* ignores this percept.

The primary action that a bot can send to the environment is `play_card(N)`, where `N` is the index position of the card in the bot's hand that it wishes to play. This action is executed after the bot has made its decision based on the percepts received from the environment. The `BriscolaEnvironment` processes this action by telling the GUI that the bot played a specific card, then the GUI updates the game state to reflect the card played by the bot and continuing the flow of the game.

5 Implementation Details

For detailed documentation, it has been generated the **KDoc** (the equivalent of Java's Javadoc) for the implemented classes and methods, available on the GitHub page[2]. This documentation provides in-depth insights into the various components and their interactions, offering a complete understanding of the system's architecture and implementation.

6 Self-assessment / Validation

During the development, a series of comprehensive unit tests were conducted to ensure the correct functioning of each implemented feature. Each functionality was individually tested, observing its behavior in isolation whenever possible. Subsequently, once integrated with the rest of the system, further testing was carried out to evaluate the overall behavior of the game.

Evaluation Criteria The evaluation of the produced software was based on its compliance with the following criteria:

- **Functional Requirements Compliance**: Each feature was tested against the specified functional requirements to ensure it met the desired functionality.
- **Reliability and Robustness**: The system's ability to handle various conditions without failure was evaluated through stress testing.
- **Performance**: The responsiveness and speed of the system were assessed during game play.
- **User Interface Quality**: The usability and intuitiveness of the GUI were evaluated through manual testing.

Testing strategy Extensive unit tests were implemented using Kotlin Test for the `utils` and `model` packages, ensuring that each component performed as expected. Integration tests were conducted to verify the interaction between different components.

Test Coverage and Results

- **Total Number of Tests**: 33
- **Passing Tests**: 33
- **Test Coverage**: 92%

The high test coverage ensured that most of the codebase was covered by tests, providing a strong guarantee of reliability.

System Testing During system testing, it was crucial to use the debugging features provided by the MAS platform. This allowed monitoring the current state of beliefs, plans, and actions of the agents at specific moments in the game. This support was essential for identifying any anomalies in the behavior of the bots and for optimizing the application.

Identifying and Resolving Issues Some issues encountered during testing were difficult to identify and understand due to the randomness of deck shuffling, making it challenging to consistently reproduce specific scenarios. Extensive observations and numerous simulation tests were conducted to verify the bots' decisions at different intelligence levels. This process required patience and determination but ultimately led to problem resolution and overall improvement of the game.

Additionally, the use of CI/CD with GitHub Actions facilitated automatic testing, helping to quickly identify and address issues, ensuring the reliability and robustness of the system.

7 Deployment Instructions

To install and launch the Intelligent Briscola game, follow these steps:

1. Clone the repository:

   ```
   git clone https://github.com/andreazammarchi3/intelligent-briscola.git
   ```

2. Navigate to the project directory:

   ```
   cd intelligent-briscola
   ```

3. Install dependencies and build the project. Ensure that you have Gradle installed. Once Gradle is installed, run:

   ```
   ./gradlew build
   ```

4. Run the application:

   ```
   ./gradlew run
   ```

These steps will set up the environment, install necessary dependencies, and start the Intelligent Briscola game. Enjoy playing!

Figure 14: Menu View

8 Usage Examples

Figure 14 shows the home menu that appears once the app has been launched. Here the player can enter his name, select the bot's level of intelligence and choose who starts first. After that he can start a new match.

Figure 15 shows the match view that appears once the match started. Here the player can challenge the bot by adopting the strategy he deems most appropriate. The player can only play a single card when it's his turn or decide to quit the match and go back to the home menu.

Figure 16 shows the scoreboard view that can be reached from the home menu. Here the player can see all the previous ended matches that he or another player made. The columns are sortable so the player can see the list of the matches ordered by player name, by bot level, by result (win, loss or draw), by player points or by bot points.

9 Conclusions

In this project, I developed the Intelligent Briscola game, a digital version of the classic Italian card game featuring bots with varying levels of intelligence. The system was designed using a multi-agent framework with Kotlin for the core logic, Jason for agent behaviors, and JavaFX for the user interface. I implemented extensive testing procedures, to ensure the reliability and robustness of the game. Additionally, I utilized CI/CD pipelines with GitHub Actions for automatic testing, which streamlined

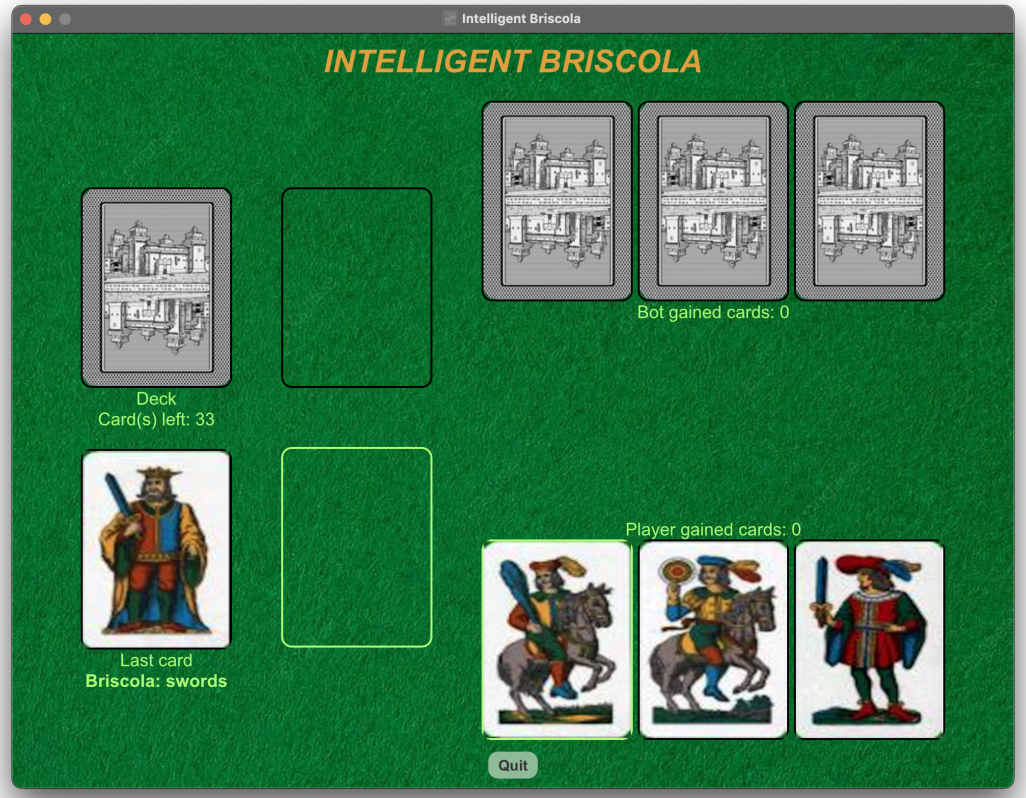

Figure 15: Match View

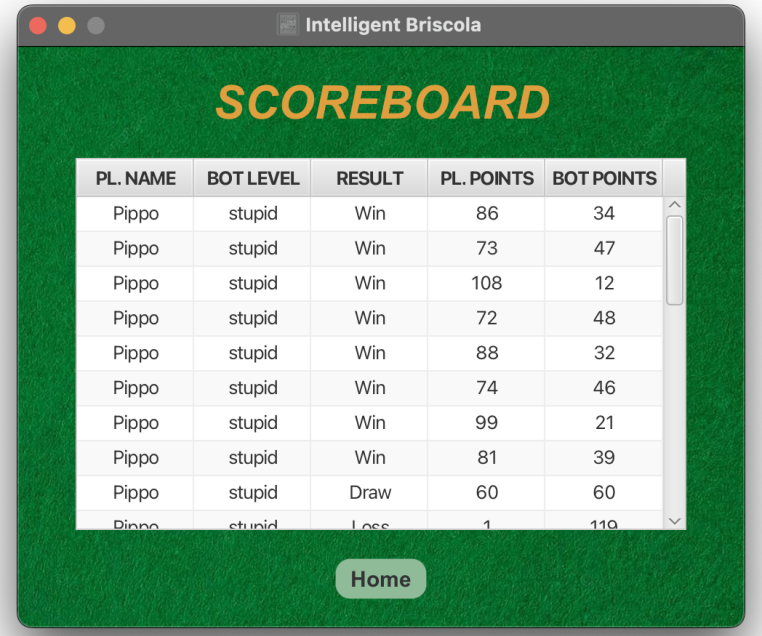

Figure 16: Scoreboard View

the development and testing process.

9.1 Future Works

Despite the comprehensive development and testing efforts, there are several areas that were not fully explored and can be addressed in future work. These include:

- **Enhanced AI Strategies**: Developing more advanced and diverse strategies for the intelligent bot to provide a greater challenge for players. For example, strategies based on the previous played cards.
- **2v2 Support**: Introducing a mode to allow players to compete against each other in the famous 2v2 formula.

9.2 What did we learned

Throughout this project, I gained valuable insights into the development of a complex, multi-agent system. Key learnings include:

- **Multi-Agent Systems**: Understanding the intricacies of coordinating agents using Jason, ensuring their actions align with the overall game logic and also understand their influence on a custom environment.
- **AI Strategy Implementation**: Developing and refining AI strategies to create bots that offer varying levels of challenge to players.

References

[1] Wikipedia. Briscola official rules. `https://en.wikipedia.org/wiki/Briscola`. Accessed: 2024-06-19.

[2] Andrea Zammarchi. Intelligent briscola: The classic card game, with the possibility to play against an intelligent bot. `https://andreazammarchi3.github.io/intelligent-briscola/`, 2024. Accessed: 2024-06-19.