OmokGame

Fecha de creación	@8 de noviembre de 2023 17:04
Clase	Advanced Object Oriented Programing
Tipo	All
Revisado	☐

Starting the Game:

- When the game starts, the `Main` class initializes `SelectionMenu` for the player to choose between playing against another human or AI.
- After the selection, `BoardFrame` is instantiated, creating the main game window. It initializes the `OmokGame` class, passing in the players and whether an AI is involved.

Player Initialization:

- In `OmokGame`, `player1` is always a human player, while `player2` is either another human or an AI (`AIPlayer`), depending on the choice made in `SelectionMenu`.
- `player1Panel` and `player2Panel` are initialized to display player information, such as names and statuses.

Gameplay:

- The `OmokGame` class listens for mouse clicks on the `Board` panel. When a player clicks, it translates the click into board coordinates and attempts to place a stone through the `makeMove` method.
- If the position is already occupied, the game notifies the player and asks for another move.
- Once a move is made, the game checks if it results in a win or a draw. If so, it displays a message and sets the game state to over.
- If the game is not over, it switches to the other player.

AI Moves:

- If the current player is the AI (`AIPlayer`), `OmokGame` uses `SwingUtilities.invokeLater` to make the AI's move in a thread-safe manner.
- The AI computes its move using the `bestMove` method, which employs a MiniMax algorithm with alpha-beta pruning to find the optimal move.
- After the AI makes its move, the game checks for a win or a draw, just like after a human player's move.

User Interface Interactions:

- The `BoardFrame` includes a menu bar and tool bar for actions like starting a new game, changing usernames, or quitting to the selection menu.
- The `PlayerPanel` class updates to show the current player's status, animating dots to indicate waiting for a move and highlighting the panel of the player whose turn it is.

Game State Management:

- The `Board` class manages the state of the game board, storing the positions of the stones and evaluating the board for win conditions.
- `Player` objects maintain the last move made by each player and their stone type, with `HumanPlayer` and `AIPlayer` handling the specifics of move-making.

Ending and Resetting the Game:

- `OmokGame` provides functionality to reset the game to its initial state, clearing the board and starting anew with `player1`.
- Upon winning, losing, or drawing, the game presents the outcome to the players and stops further moves until a reset occurs.

Overall Behavior:

- The game operates in a turn-based manner, alternating between two players, with one of the players possibly being an AI.
- The GUI is responsive and interactive, providing real-time feedback on the game's progress and player actions.
- The game logic is encapsulated in the `OmokGame` class, which orchestrates the flow of the game, delegating to `Board` for board state management and to `Player` subclasses for move-making logic.

- `AIPlayer` uses strategic AI to challenge the human player(s), making the game engaging and competitive.

This comprehensive overview illustrates a well-organized structure of the game, with clear separation of concerns between the user interface, game logic, and AI decision-making, resulting in a functional and interactive Omok game application.

OmokGame: This is the main game class which controls the flow of the game. It has associations with `Player` objects, indicating that it manages players. It also has a boolean that might be used to keep track of the game state (e.g., whether the game is currently active). Yet Main Class is the main call of the Whole Game

1. **Player**: This abstract class could represent a generic player in the game. It has attributes like `name`, `stoneType`, and `lastMove` to keep track of each player's details and actions. There are two concrete subclasses:
    - **AIPlayer**: Represents a computer-controlled player, with methods for choosing moves based on the game state (`bestMove()`, `countConsecutiveStone()`, etc.).
    - **HumanPlayer**: Represents a human player, with methods for making moves based on user input (`makeMove()`).
2. **Stone**: This class likely represents the stones that players place on the board, with attributes to identify the type of stone and possibly other gameplay-related properties.
3. **Board**: Represents the game board. It has methods to manipulate and check the board's state, such as `checkDirection()`, `isEmpty()`, and `isFull()`.
4. **BoardFrame** and **SelectionMenu**: These classes seem to be part of the user interface, with `BoardFrame` likely being the main window of the game and `SelectionMenu` being a component where players can make selections (probably of game mode or difficulty).
5. **ImageHandler**: This class probably handles loading and manipulating images, which could be used for visual representations of the board, stones, and other graphical elements.
6. **Main**: This class contains the `main()` method, which is the entry point of the application. It seems to create the game's GUI and start the game.

Overall Behaivor:

- The application is started via the `Main` class, which sets up the GUI and initializes the game.
- The `OmokGame` class is instantiated, setting up the necessary `Player` objects (which could be `AIPlayer` or `HumanPlayer`).
- Players take turns making moves, with `HumanPlayer` waiting for user input and `AIPlayer` calculating the best move.
- The `Board` class keeps track of the game state and validates moves.
- The `BoardFrame` updates the visual representation of the board after each move.
- The `ImageHandler` provides the necessary images for the `BoardFrame`.
- The game continues until the `Board` is full or a player wins, as determined by the game logic within `OmokGame`.