Final Documentation - Code Explanation

Game Flow Overview

The game starts with the MainMenu, where players can select the number of participants, choose whether each one is human or AI, set their names and assign colors. After the setup, control is passed to the MainGameWindow, which initializes the full graphical interface. This includes:

- GameGrid, a 7x7 grid representing the board;
- DicePanel, which shows six dice and allows for rolling and re-rolling;
- PlayerPanel and GameStatisticsPanel, which display player-specific and general game information.

The game progresses over 10 rounds. In each round, every player gets one turn. During a turn, a player can roll the dice up to three times, choosing which dice to re-roll. If the final combination matches a pattern required by any board cell, the player can place a stone there. After all players have completed a round, the game advances to the next one. Once all rounds are completed, GameManager evaluates the scores, and the player with the highest total wins.

Class Responsibilities

Each class in the application serves a distinct role, contributing to a well-organized and maintainable codebase:

- **MainMenu**: Handles game setup, including player selection and configurations. Once ready, it launches the main game interface.
- **MainGameWindow**: Acts as the central controller by coordinating GUI elements and handling the overall game logic. It follows the GRASP Controller pattern.
- **GameManager**: Maintains core gameplay logic, such as turn management, score updates, and round transitions. It also creates players and the board, making it both an Information Expert and a Creator.
- **Board**: Represents the underlying 7x7 grid model, storing the state of each cell and providing methods for stone placement.

- **GameGrid**: Offers the visual representation of the board and highlights cells that meet the conditions for valid moves. It contains the logic to verify dice combinations for each cell, which supports the Protected Variations pattern.
- **DicePanel**: Handles all dice-related functions, such as generating random values, managing re-rolls, and updating the display based on user actions.
- **PlayerPanel** and **GameStatisticsPanel**: Display dynamic information, such as current scores, player turns, and remaining stones, though they do not contain core logic.
- **Player**: Stores each player's name, score, stone color, and whether they are human or AI. It acts as the Information Expert for its own state.
- **ComputerPlayerAI**: Controls AI behavior, from rolling dice to placing stones. It is considered a Pure Fabrication as it doesn't represent a domain concept but isolates the AI logic.
- **DotIcon** and **DiceComponent**: Handle rendering of stones and dice visually. These utility classes separate visual concerns from logic.

User Interaction

Players interact with the game through buttons and grid cells. When a player clicks the "Roll" button in the DicePanel, dice values are randomly generated and displayed. The player can then choose specific dice to hold and re-roll the others. The GameGrid checks the current dice values against the requirements of each board cell using the validateDiceCombination() method and highlights all valid targets.

If the player clicks a highlighted cell, a listener (GridCellClickListener) relays the action to the GameManager, which verifies the move. If the move is valid, the player's stone is placed on the board, their score is updated, and the UI reflects these changes. If the move is invalid (such as clicking an already occupied cell), no change is made.

After each turn, the control shifts to the next player. The interface updates to show the new player's turn, and only that player can interact with the interface. This separation of input and logic, handled through dedicated listeners and coordinated by the controller, ensures the interface remains responsive and the logic remains robust.

Dice and Grid Logic

The DicePanel maintains the values of all six dice after each roll. These values are used to evaluate whether any move on the board is possible. The GameGrid contains logic to evaluate specific patterns such as "all even", "sum greater than 30", "three pairs", or "large straight". These validations are encapsulated in the method validateDiceCombination(). When dice are rolled, the GameGrid uses this method to check each board cell's condition against the rolled dice. Only those cells that match are highlighted for interaction. This approach centralizes rule-checking and makes future updates to the rules easy, since changes are limited to this one method. This implementation reflects the GRASP pattern of Protected Variations by shielding the rest of the system from changes in rule logic.

By separating dice rolling (random behavior) and rule checking (logical evaluation), the system maintains clear cohesion and low coupling, both of which are principles of well-designed software architecture.

AI Player Logic

When it is the AI player's turn, the ComputerPlayerAI class takes over the entire decision-making process. The AI rolls dice (up to three times), just like a human player. After each roll, it uses the same logic as the human player to check valid board positions by calling validateDiceCombination().

If there are valid cells available, the AI selects the best one to place its stone and this might be the first valid option found or the highest-scoring cell depending on the strategy used. If the AI fails to find a valid move after all three rolls, it will skip the turn. Once the move is made or skipped, the AI ends its turn and signals the GameManager to proceed to the next player.

To prevent the UI from freezing during AI actions, the logic runs on a separate thread. This ensures a smooth user experience. Currently, the AI logic uses basic conditional checks, but it could be improved by introducing polymorphism. For example, having EasyAI and HardAI classes implement as a shared AIStrategy interface.

Save and Load Functionality

The game supports saving and resuming sessions using Java's object serialization. When the user chooses to save the game, the current GameManager object (along with the list of players, the board, and other internal states) is written to a file. Since GameManager holds all critical data, serializing it captures the complete state.

To load a game, the file is read and the saved GameManager object is deserialized. A new MainGameWindow is launched using this restored game state. All necessary classes involved in the game state are marked Serializable, which allows Java's built-in serialization mechanism to handle complex object graphs with ease.

Transient states like AI threads are not serialized and are restarted when needed. This separation ensures that runtime behaviors are not entangled with persistent game data. It also aligns with the principle of high cohesion: each class remains focused on its primary role, whether it be game logic or persistence handling.