**Domain model**

**Graphical user interface, application, Teams

Description automatically generated**

**CVA scheme**

|  |  |
| --- | --- |
| Commonality | Variations |
| Board | 8 holes, 6 holes, n holes |
| Stone | White stones, black holes |
| Starting N.O. stones | 4, 6, n |
| Rules for win | Ruleset 1, 2 3 |
| Rules for move | Ruleset 1, 2 3 |

**Analysis matrix**

|  |  |  |  |
| --- | --- | --- | --- |
| Use-case | Ruleset 1 | Ruleset 2 | Ruleset 3 |
| Score is calculated | Score equals to number of stones in Basehole |  |  |
| Determine winner | Player with highest score wins |  |  |
| Perform next move | Spread the stones in selected hole over the next (CCW) holes, where every hole receives one stone |  |  |

The number of stones, holes and color of said stones are not determined by the ruleset but by the player and can be changed per game. Thus, it is not included in the analysis matrix.

In our class diagram we have chosen to apply a strategy pattern to IRuleSet, the ruleset of the game. This way we allow the Game class to use RuleSet methods without depending on a specific implementation. This helps by ensuring that the Game class can use RuleSet methods for different sets of rules. We believe this to be the best pattern to use because we want to use different variants of an algorithm within an object and be able to switch from one algorithm to another during runtime. This prevents hard-wiring of all the algorithms into the program.

We use the helper class Boardstate to prevent us from hard coding too much logic. While this is not a design pattern, we do feel the need to explain this choice. This method helps reduce code coupling by taking stress off the link between IRuleset and board. This way we keep our code neat, efficient and future proof.

* Game
  + This class is responsible for storing the board, the rules, 2 players, starting a new game, and making new moves
* Player
  + This class is responsible for storing the player values color, score, and isAI
* Board
  + This class is responsible for the amount of stones in each hole, getting the boardstate, and printing a representation of the current board to the screen
* Boardstate
  + This class is responsible for using the rules to determine the state of the board
* IRuleSet
  + This interface is responsible for the starting values of the amount of holes per player, the amount of starting stones per hole, checking the validity of a move and determining the winner of a game
* RuleSet1
  + This is an implementation of a RuleSet
* RuleSet2
  + This is an implementation of a RuleSet