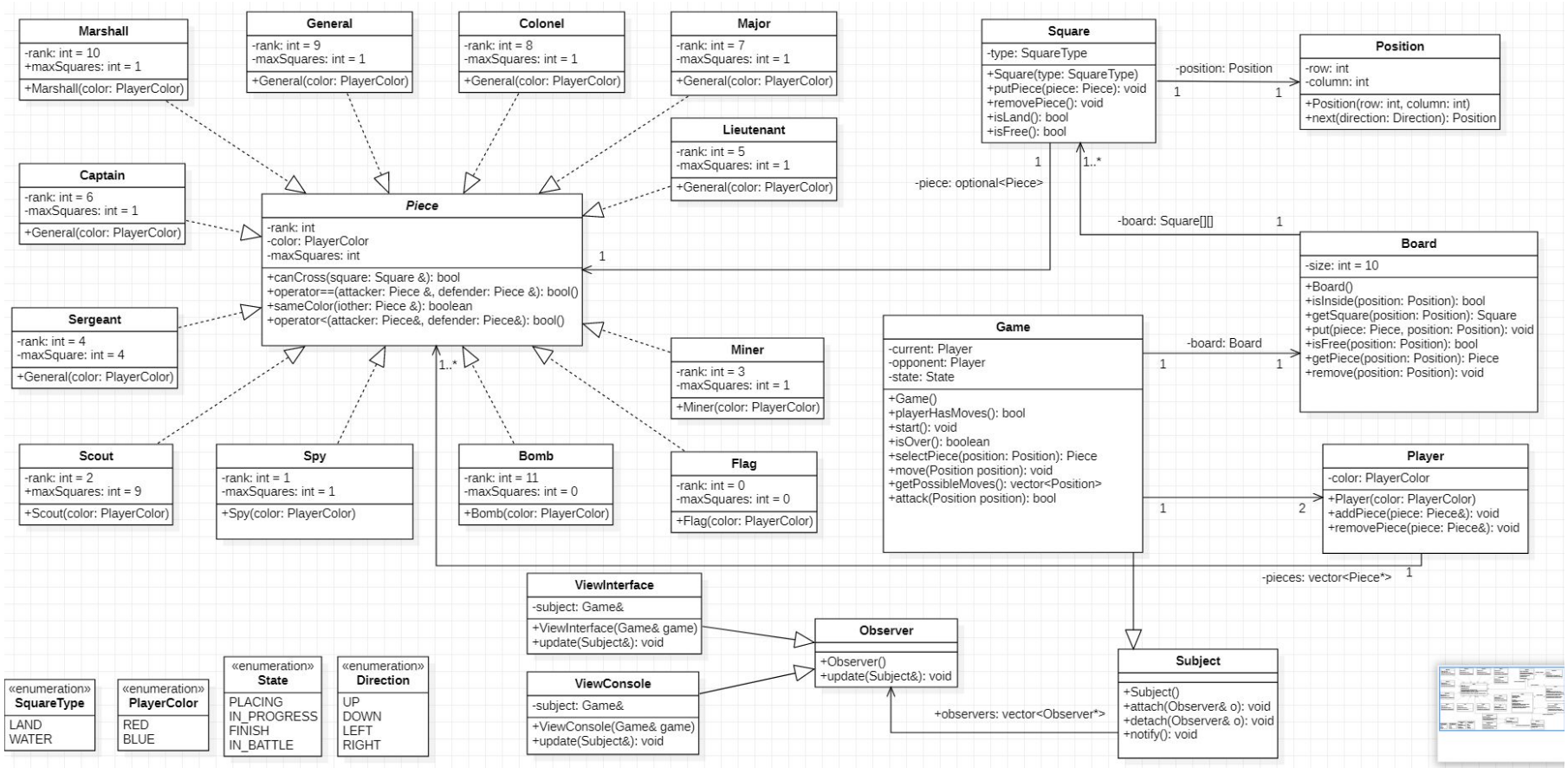
STRATEGO

Participants:
Gracziela Sewodo, 52949
Iuliana Costil, 55994

Professor:
Marcelo Burda

Analyse of our project



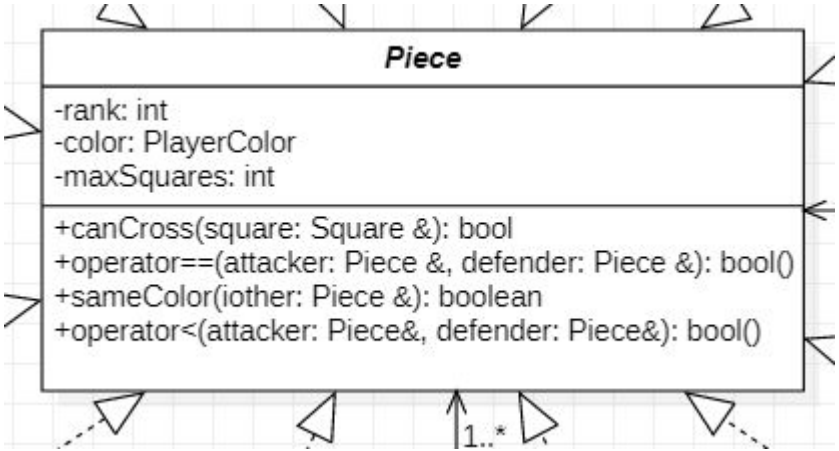
Piece: serves as an abstract interface that will determine the behavior of our different pieces

Attributes:

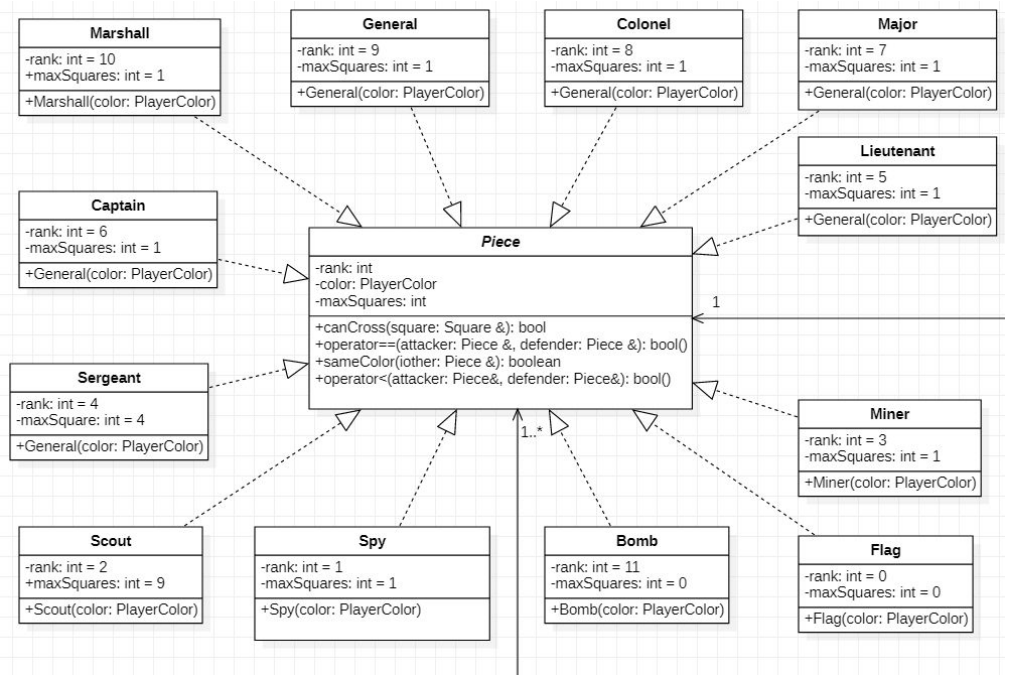
- rank: integer that represents the rank of the piece
- color: the color of the player that owns the piece
- maxSquares: integer, the maximum of squares that the piece can cross

Methods

- Piece(in rank:int, in color:Color, in maxSteps:int) : constructor used to create a piece
- canCross(in square:Square)-> boolean : provides a boolean that shows if the piece can cross a certain square
- operator<(currentPiece: Piece,other: Piece): provides a boolean that is true if the rankof the first piece is lower than the second, false otherwise
- operator==(currentPiece: Piece,other: Piece): provides a boolean that shows is if the pieces have the same rank
- same_color(other:Piece)-> boolean : provides a boolean that shows if the pieces have the same color



The different types of pieces are implemented in 12 classes



There are multiple classes with the same behavior, but have particular implementation. For illustration purposes, we can divide them in special and simple ones:

Special	**Simple**
Marshall	Captain
Spy	Sergeant
Scout	General
Miner	Colonel
Bomb	Major
Flag	Lieutenant

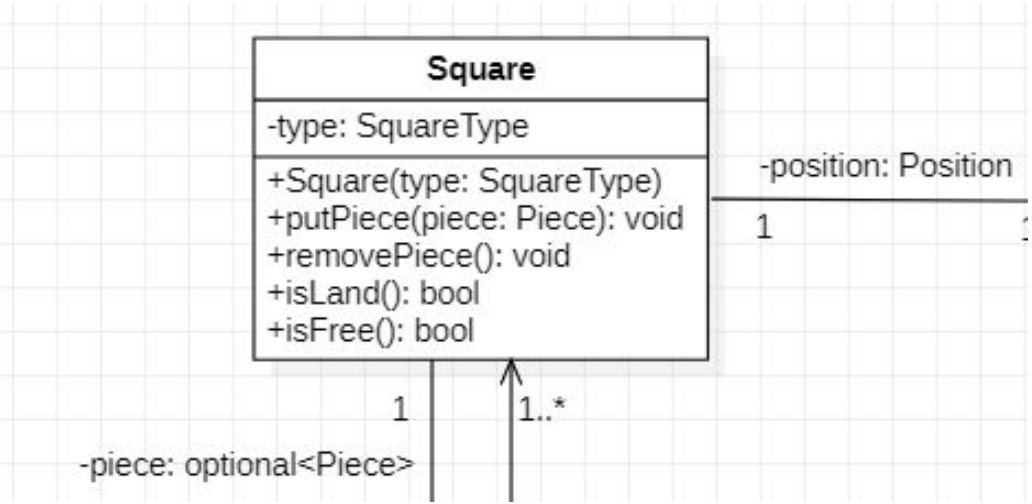
Square: refers to a square of our 10x10 squares game board

Attributes:

- type: the type of the square (LAND or Water)
- piece: represents a piece in the square
- position: the Position where the square situates

Methods:

- Square(type: SquareType) : constructor used to create a square
- putPiece(piece: Piece) : add a piece in the square
- removePiece() : remove the piece of the square
- isLand(): provides a boolean that is true if the square type is LAND, false otherwise



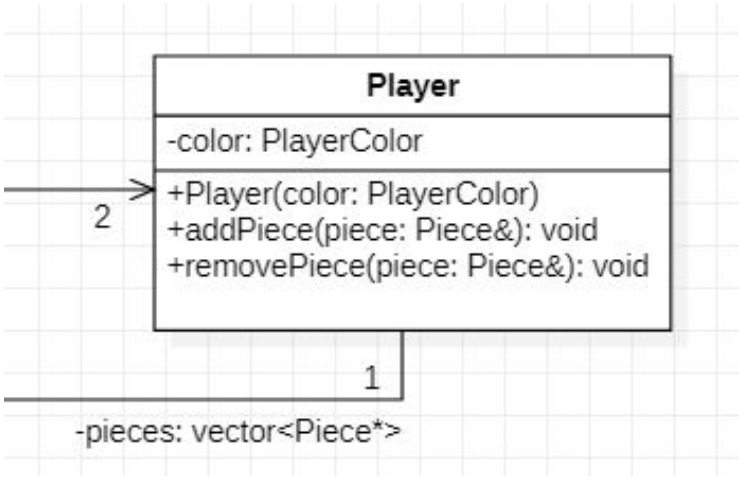
Player: the user that will play the game

Attributes:

- color: the color of the player (BLUE or RED)
- pieces: List that represents all the pieces of the player

Methods:

- Player(color: PlayerColor) : constructor used to create a player
- addPiece(piece: Piece) : add a piece to the player's pieces
- removePiece(piece: PIece) : remove the piece given to the player's pieces



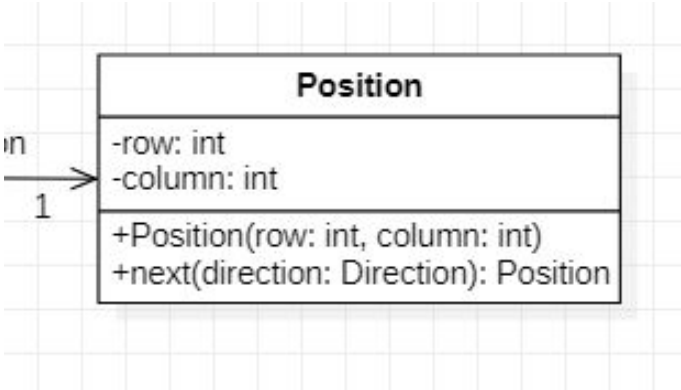
Position: represents a position in the game board

Attributes:

- row: integer represents a row
- column: integer represent a column

Methods:

- Position(row: int, column: int) : constructor used to create a Position
- next(direction:Direction) ->Position: Give the next position in the given direction
- getRow()->int : getter of row
- getColumn() ->int : getter of column



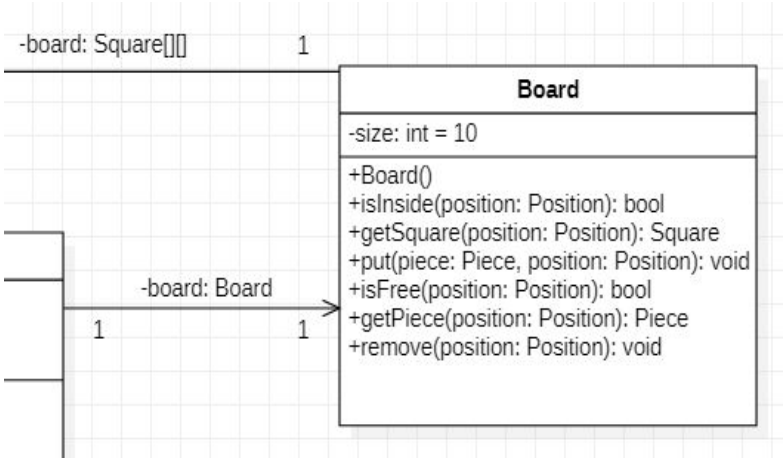
Board: represents the board of the game, that will contain our pieces

Attributes:

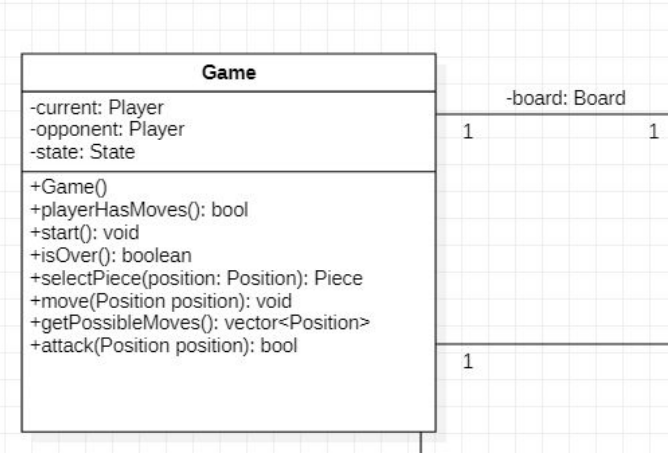
- row: integer the number of rows
- column: integer the number of columns
- board: Square[][] a 2D array of squares

Methods:

- Board() : constructor of Board
- isInside(position:Position)->boolean: provides a boolean that is true if the position isinside the Board , false otherwise
- getSquare(in position:Position)-> Square: return the square of the given position
- put(piece:Piece, in position:Position) : put the given piece to the square at the given position
- isFree(in position:Position)-> boolean : provides a boolean that is true if the the square at the position is free , false otherwise
- getPiece(position:Position)-> Piece: return the piece who is at the square at the given position
- remove(position:Position): . Remove the piece of the square at the position given



Game: the class that will structure our project joining all the other classes together



Attributes

- board: a Board is the container of our pieces
- current: Player that is currently playing
- opponent: Player that is the adversary of the current player
- state: the current State of our game

Methods

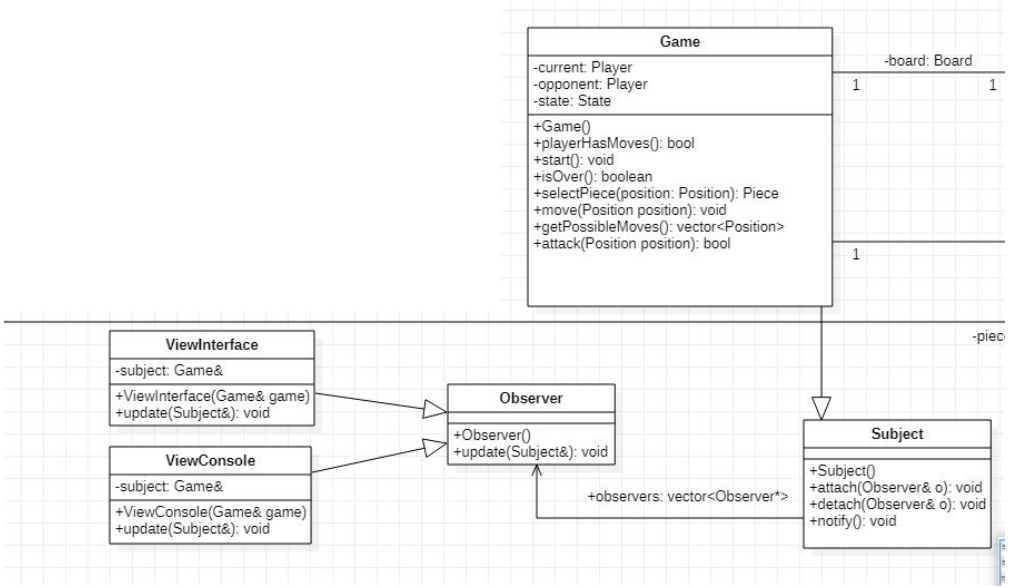
- Game(): create a new game with default setting
- has_moves(in player:Player): provides boolean the indicates if the current player still has possible moves
- start(): method used to setup the game
- is_over(): provides boolean that indicates if the game is over
- select(in row:int, in column:int): select the piece you want to play with
- get_selected(): provides the Piece selected by the current player
- move(in piece:Piece, in direction:Direction): moves the selected Piece in the wanted direction
- battle(in position:Position):
- getMoves(in piece:Piece) provides all the possinle moves that can be done by the selected piece

Observer Pattern

We will use the Observer pattern in order to be able to attach 2 types of views to our subject (Game).

The following classes are added to our project:

- Observer : respresent the behavior of our views
- ViewConsole
- ViewInterface
- Subject : represents the behavior of our game



The enumerations that we will be using

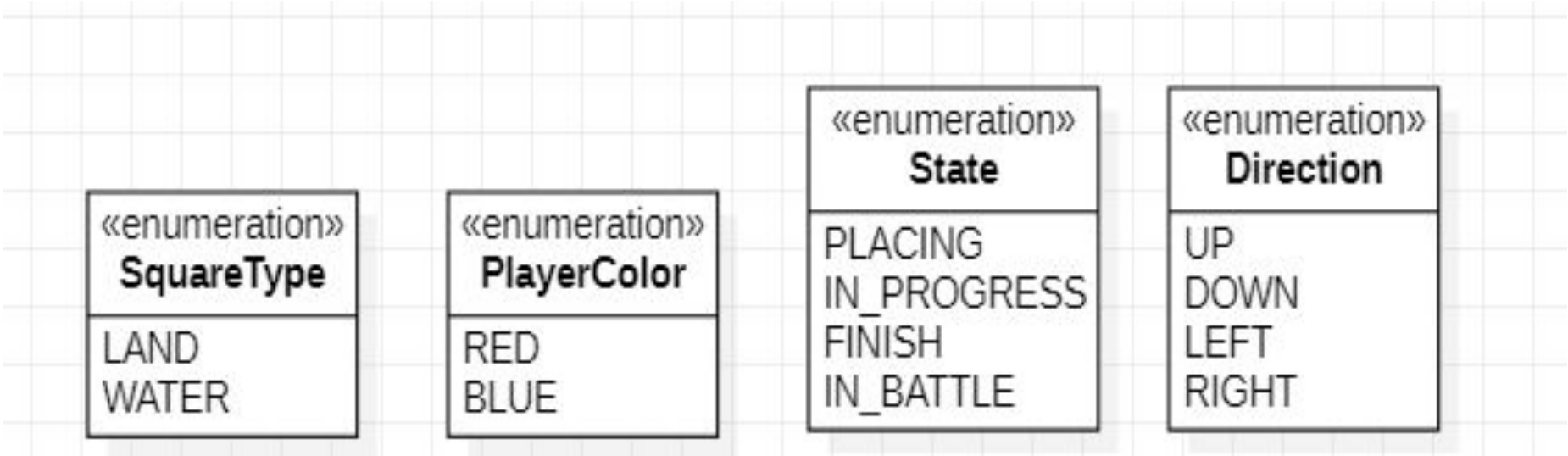


diagram v2 console

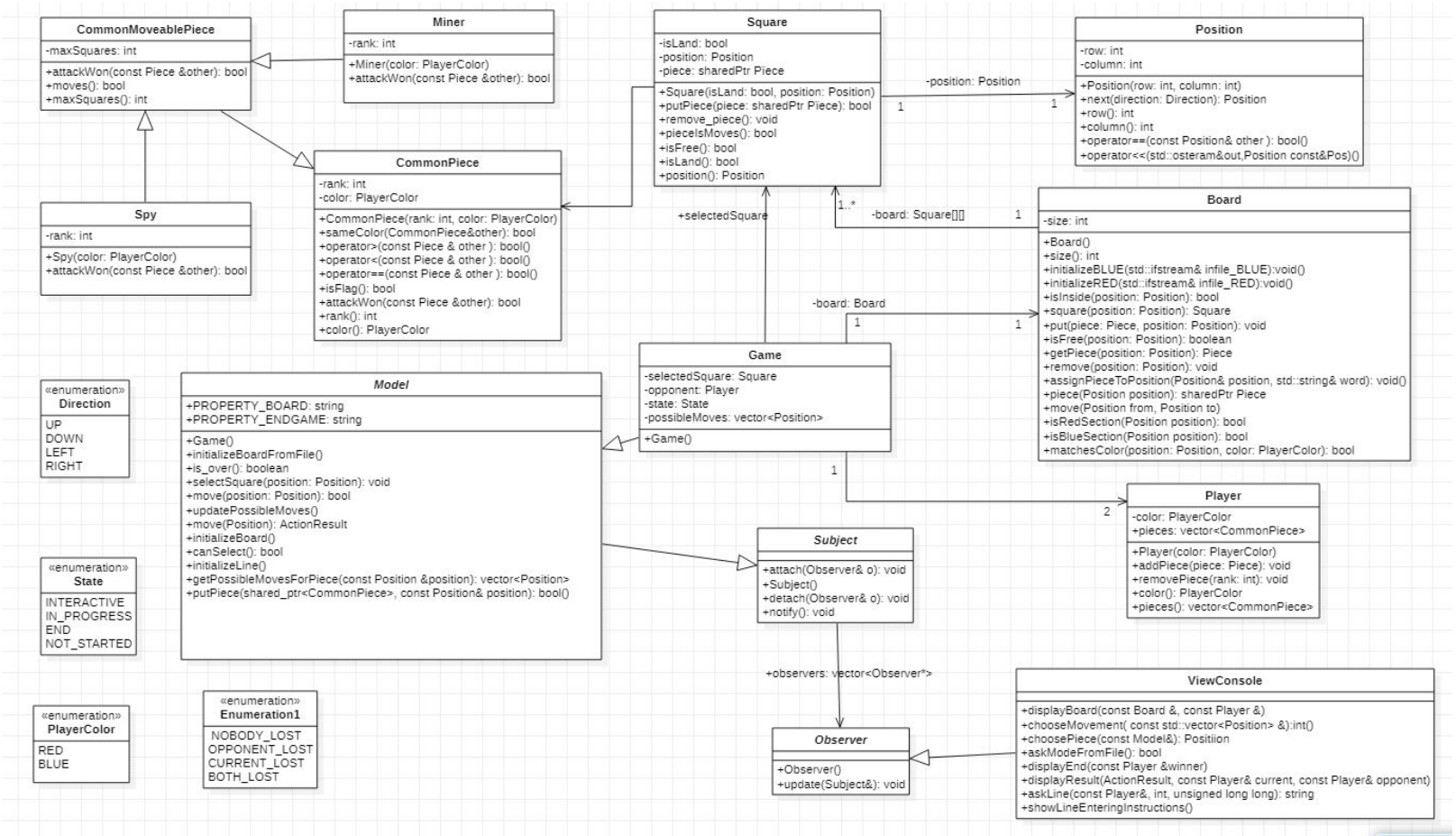


diagram v3 gui

Changes :

Model class:

-we added a new attributes “ActionResult currentResult_ “ that gives us the result of the last move

- move method: the method return is now void. we update the new currentResult in the method

diagram v3 gui