

8		ROOK		KNIGHT		BISHOP		QUEEN		KING		BISHOP		KNIGHT		ROOK	
7		PAWN		PAWN		PAWN		PAWN		PAWN		PAWN		PAWN		PAWN	
6		-----		-----		-----		-----		-----		-----		-----		-----	
5		-----		-----		-----		-----		-----		-----		-----		-----	
4		-----		-----		-----		-----		-----		-----		-----		-----	
3		-----		-----		-----		-----		-----		-----		-----		-----	
2		PAWN		PAWN		PAWN		PAWN		PAWN		PAWN		PAWN		PAWN	
1		ROOK		KNIGHT		BISHOP		QUEEN		KING		BISHOP		KNIGHT		ROOK	
		a		b		c		d		e		f		g		h	

Chess Game

Malek Al-Sadi

Software Engineering
JUST

Topics

1- About chess

- What are the pieces
- What is the goal for each player

2- Main classes

- chessGame
- chessboard
- Square
- Piece
- Validation
- checkMate

3- Design Patterns

4- Solid Principles

1- *About chess*

a. What are the Pieces?

The game contains of board contains from black and white squares and 32 pieces. Pieces in the game are divided into 2 identical groups each group for a player initially in a side of the board, which mean it's 1v1 game, one of them contains white colored pieces and the other are black, each group have 6 deferent Piece each with different type of move

1. Rook

That only moves in straight clear of Pieces lines

2. Knight

That only walk in road like the 'L' letter, and the road he walked through doesn't have to be clear of Piece, he's able to jump.

3. Bishop

That only moves in diagonally clear of Piece lines

4. Queen

Can move both straight and diagonally ways but, these roads must be clear of Pieces

5. King

Like the heart of the body, when the King is down, it's Game Over. The King can take one step to any direction.

6. Pawn

Can only move one step towards the other side of the of board and two step if it's his first move, unlike the other pieces the pawn can kill the piece on the square he landed on, he only can kill diagonally and take that

place only if that place is toward the other side and contains other color piece.

b. What is the goal of each player?

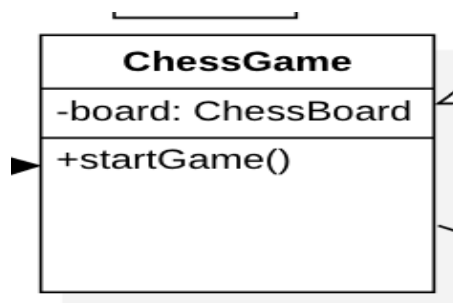
Each player is trying to corner the other one's King to be at position where whatever move he did his King will be killed

2- Main classes

There are 6 main classes in the game, some of the abstracted into others and some have other related classes

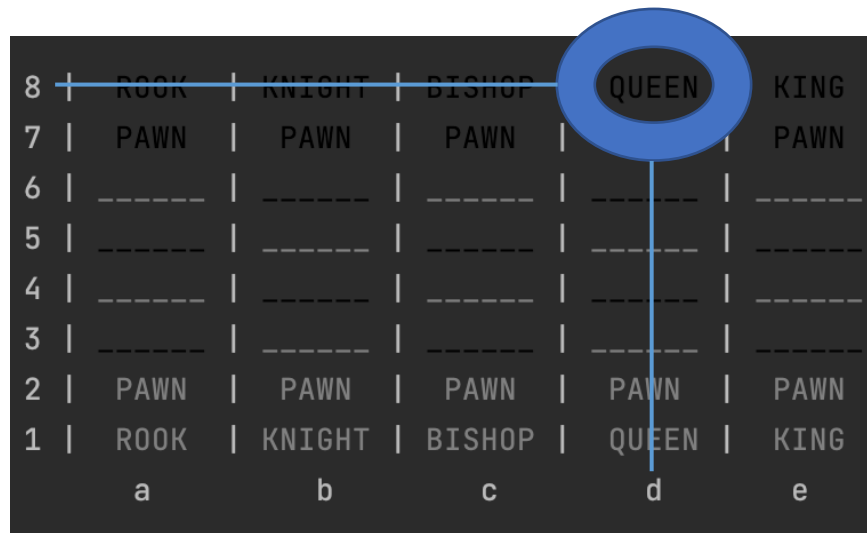
a- Chess game

Is the mediator class that controls the game and connects the game components to each other, it only contains the board that the game going to take place on and start method.



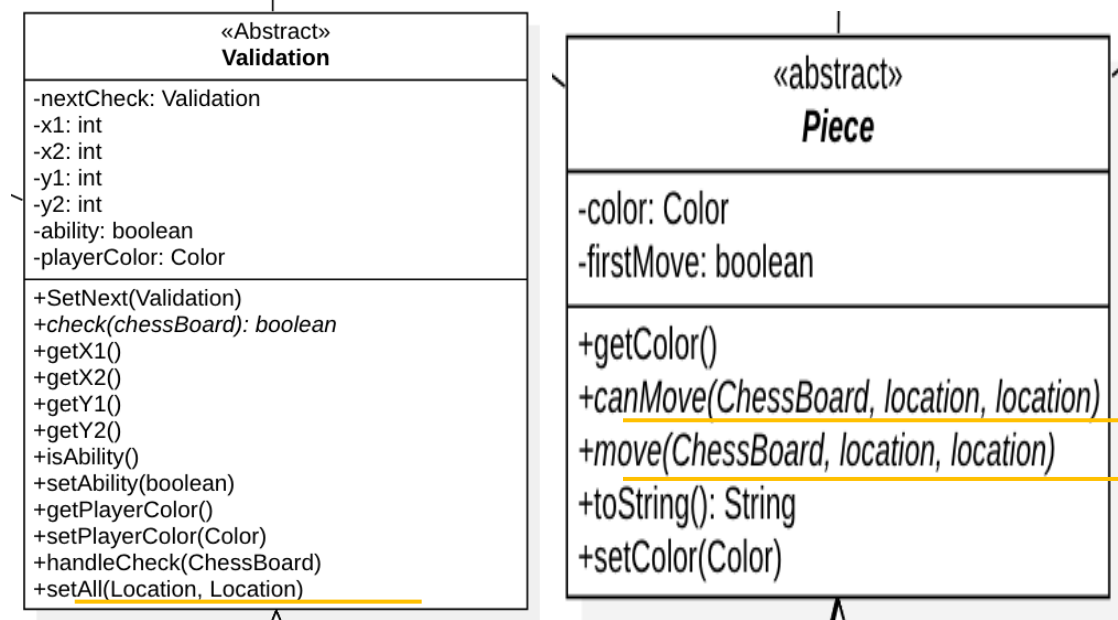
The **startGame** method prints the grid using **printgrid** method in chess board class and request the user's input. user input contains two string each with the letter and number representing the location to move from and distinction to move your piece to.

For example, Black Queen is in d8



8	ROOK	KNIGHT	BISHOP	QUEEN	KING
7	PAWN	PAWN	PAWN		PAWN
6	-----	-----	-----	-----	-----
5	-----	-----	-----	-----	-----
4	-----	-----	-----	-----	-----
3	-----	-----	-----	-----	-----
2	PAWN	PAWN	PAWN	PAWN	PAWN
1	ROOK	KNIGHT	BISHOP	QUEEN	KING
	a	b	c	d	e

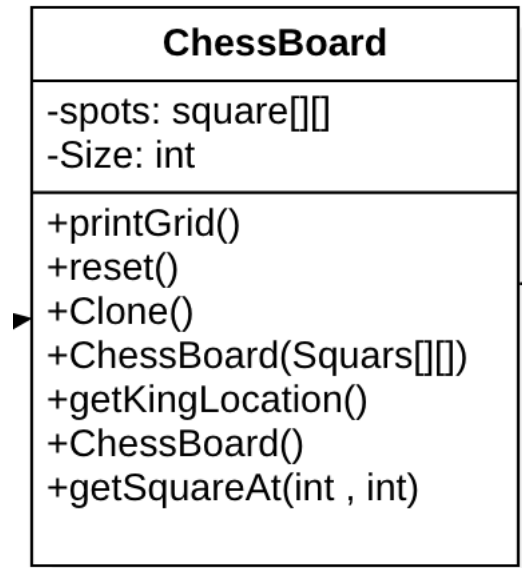
and that String is stored in object from type Location which is responsible to convert that string into x, and y axis and store them to be used in moving and validation



Then check the move validity if true perform the move otherwise notify the player and ask him to play again until performing valid move or when the player is checkmated then the game notify the players and ends the game.

b- Chessboard

Contains 2-dimensional array of Squares of size 8X8 called spots, default constructor that initialized new 2-dimensional array of Squares set them up ,and parameterized constructor that received 2d array of squares and clone each square to spots to make two copy of same chessboard



There is the **printGrid** method that print the 2D array , each element by it's overrided **toString** , before each row the row number is printed and below the grid print the character

8		ROOK		KNIGHT		BISHOP		QUEEN		KING		BISHOP		KNIGHT		ROOK	
7		PAWN		PAWN		PAWN		PAWN		PAWN		PAWN		PAWN		PAWN	
6		-----		-----		-----		-----		-----		-----		-----		-----	
5		-----		-----		-----		-----		-----		-----		-----		-----	
4		-----		-----		-----		-----		-----		-----		-----		-----	
3		-----		-----		-----		-----		-----		-----		-----		-----	
2		PAWN		PAWN		PAWN		PAWN		PAWN		PAWN		PAWN		PAWN	
1		ROOK		KNIGHT		BISHOP		QUEEN		KING		BISHOP		KNIGHT		ROOK	
		a		b		c		d		e		f		g		h	

The reset method initialized squares one by one with colors and empty piece state then gives initialized the base places for the pieces with colors, it's used to set for each new game.

```
boolean white = true;
for(int i= 0;i<Size;i++){
    for(int j=0;j<Size;j++){
        Location loc = new Location((char)(j+'a')+" "+i);
        spots[i][j] = new Square( hasPiece: false,loc);
        if(white)
            spots[i][j].setSquareColor(Color.White);
        else
            spots[i][j].setSquareColor(Color.Black);
        if(j!=Size-1)
            white = !white;
    }
}
```

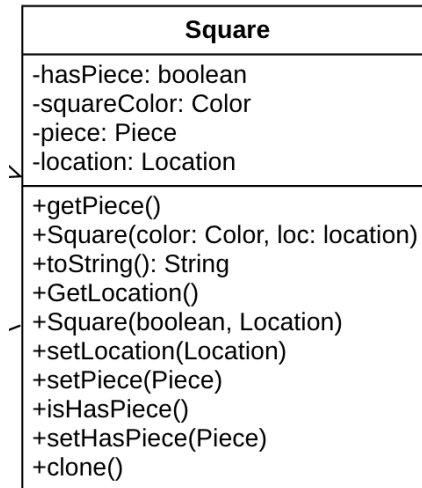
```
spots[0][0].setPiece( PieceMaker.create( kind: "Rook",Color.Black) );
spots[0][1].setPiece( PieceMaker.create( kind: "Knight",Color.Black) );
spots[0][2].setPiece( PieceMaker.create( kind: "Bishop",Color.Black) );
spots[0][3].setPiece( PieceMaker.create( kind: "Queen",Color.Black) );
spots[0][4].setPiece( PieceMaker.create( kind: "King",Color.Black) );
spots[0][5].setPiece( PieceMaker.create( kind: "Bishop",Color.Black) );
spots[0][6].setPiece( PieceMaker.create( kind: "Knight",Color.Black) );
spots[0][7].setPiece( PieceMaker.create( kind: "Rook",Color.Black) );
for(int i=0;i<Size;i++){
    spots[1][i].setPiece( PieceMaker.create( kind: "Pawn",Color.Black) );
}
for(int i=0;i<2;i++){
    for(int j=0;j<Size;j++){
        spots[i][j].setHasPiece(true);
    }
}
```

There is also **getKingsLocation** that search over all the squares to find the location of the square contains the King of the given color.

And the last important method is **getSquareAt** that receives two integer x and y and return the square at that position.

c- Square

Every square Contains the Location, Color, have piece state initially false and piece initially points to null



Contains all the typical setters and getter for these attributes and parameterized constructor that receives the hasPiece Boolean and the Location.

Contains override **clone** methods that creates new square and give it the same value of its attributes

Also contains **toString** override that return the color of the square if it doesn't contains piece or the piece's overridden **toString**

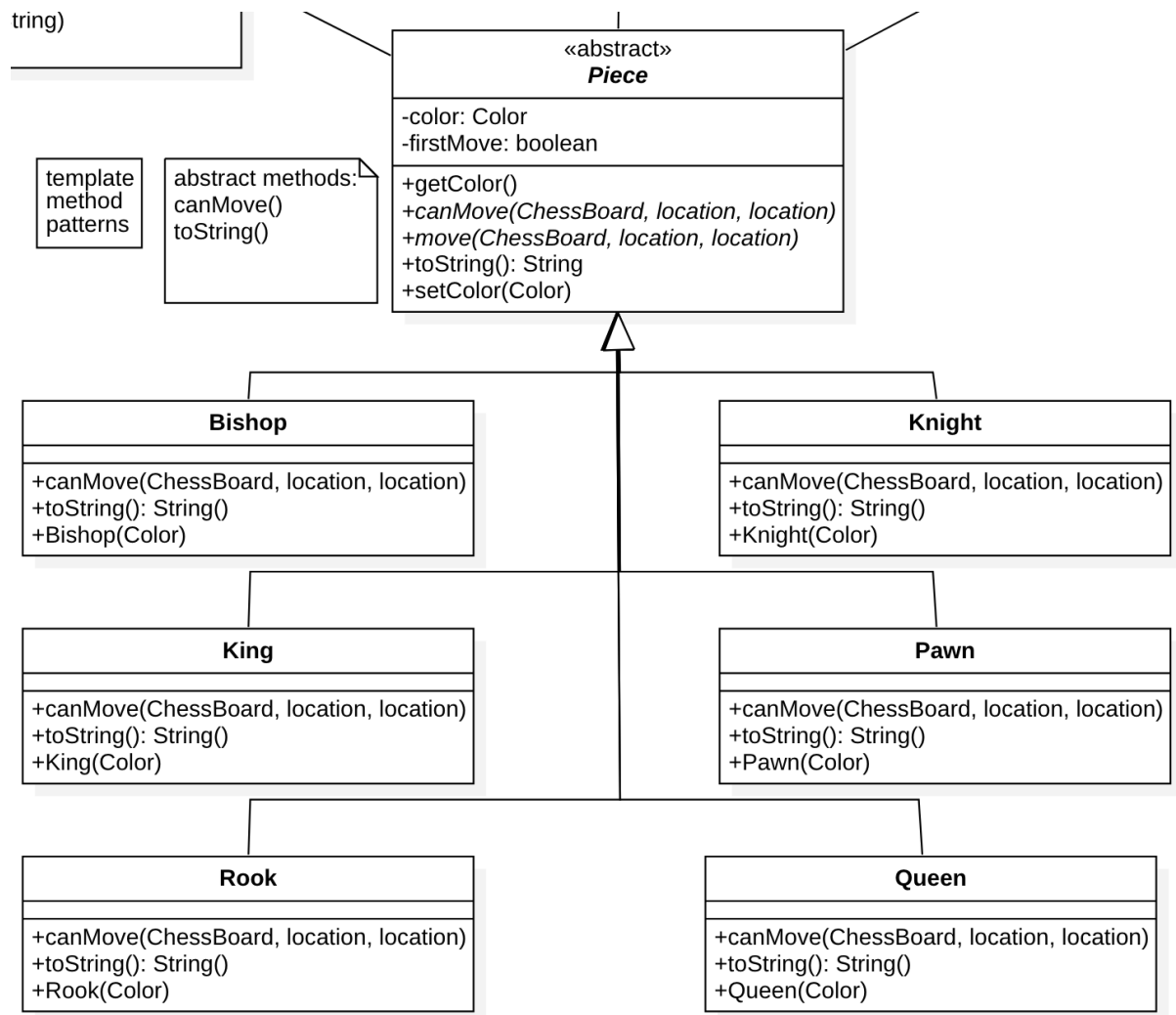
d- Piece

Abstract method that contains the color of the piece and Boolean states if it's the first move for the piece or not.

It contains the implementation of **getColor** , **setColor** method and the method responsible of changing the piece's location called **move**

That refers the end square piece to the begin square piece and the begin square piece to null.

The piece is extended in many classes repressing the mentioned above piece (Rook,Knight,Bishop,Queen,King,Pawn)



each one implements the methods **canMove** and **toString**
The can move has three main ideas

For the knight the difference of one of the dimensions must be 1 and the other must be 3 to make shape like the letter 'L'



For the Bishop the difference between the x-axis and the y-axis must be the same



For the Rook the difference between the x-axis or the y-axis must be 0 which mean moving in straight lines



For the Queen can move like the Bishop or like the Rook

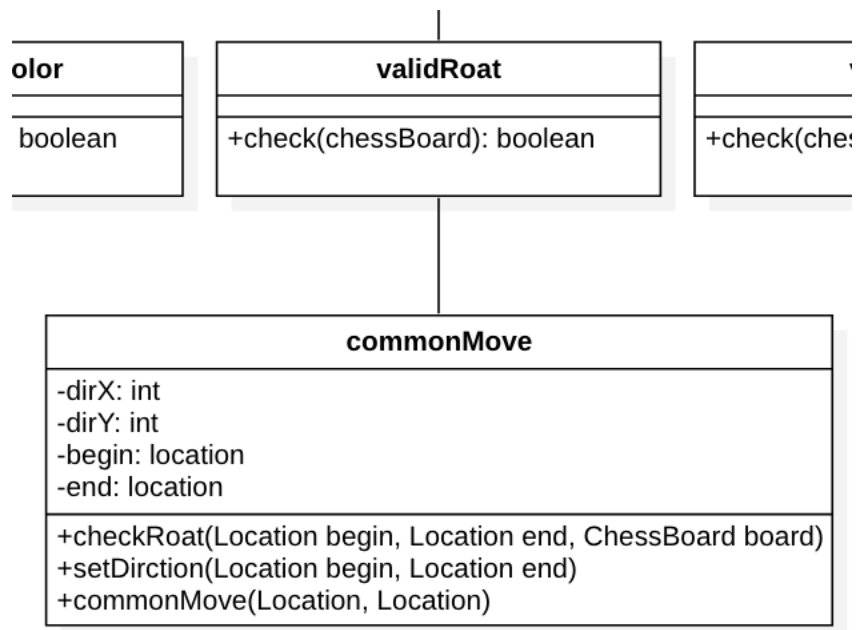
For the Pawn it can move one stop forward unless there is any piece Infront of it , it can move diagonally only if there is a piece of the other color the distention square



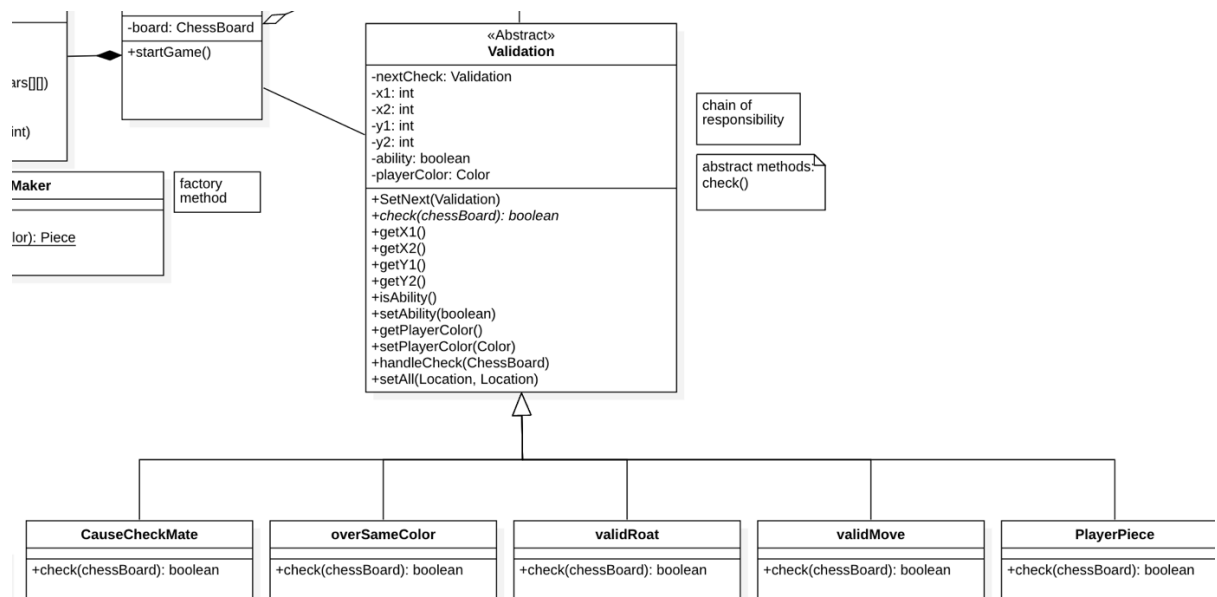
e- Validation

Abstract class the extends 5 types of validation:

- 1- playerPiece: that check if there is a piece in that square and if the piece is the same of the player color.
- 2- ValidMove: that check if the Piece move allow it to arrive the destination given
- 3- validRoat: that check if the squares in the way between the location and destination are clear of pieces using commondMoves class.



- 4- overSameColor: that check if the destination square doesn't contain piece of the same color.
- 5- CauseCheckMate: clones the chessboard and perform the move, then check all the other color pieces that no piece can move to the kings destination.



f- Checkmate

Performed at the beginning of each turn that check for all the player piece if can move anywhere in the board without causing checkmate for himself using the validation class above.

```

for(int i=0;i<8;i++){
    for(int j=0;j<8;j++){
        Square spot = board.getSquareAt(j,i);
        if(spot.isHasPiece() && spot.getPiece().getColor()==color){
            if(tryAllMoves(board,spot.getLocation(),color))
                return true;
        }
    }
}
return false;

```

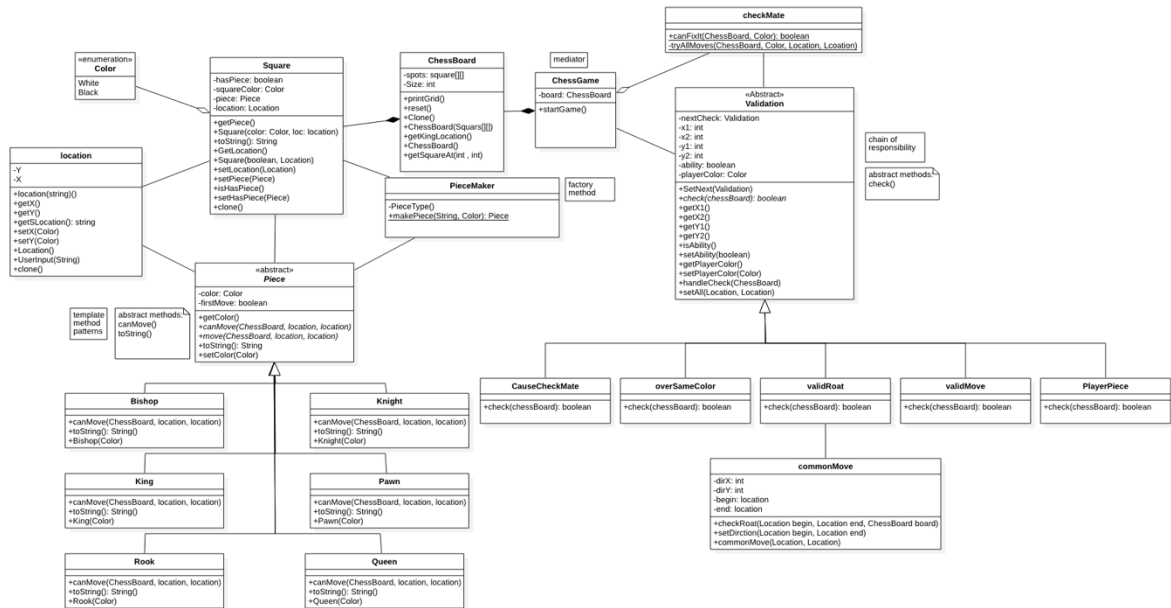
```
private static boolean tryAllMoves(CheessBoard board, Location pieceLoc, Color color){
    for(int i=0;i<8;i++){
        for(int j=0;j<8;j++){

            Location dist = new Location((char)(j+'a')+" "+i);
            Validation validChecker1 = new PlayersPiece();
            validChecker1.setAll(pieceLoc,dist);
            validChecker1.setPlayerColor(color);
            Validation validChecker2 = new validMove();
            validChecker2.setAll(pieceLoc,dist);
            Validation validChecker3 = new validRoat();
            validChecker3.setAll(pieceLoc,dist);
            Validation validChecker4 = new OverSameColor();
            validChecker4.setAll(pieceLoc,dist);
            Validation validChecker5 = new CauseCheckMate();
            validChecker5.setAll(pieceLoc,dist);
            validChecker5.setPlayerColor(color);
```

```
            validChecker1.SetNext(validChecker2);
            validChecker2.SetNext(validChecker3);
            validChecker3.SetNext(validChecker4);
            validChecker4.SetNext(validChecker5);

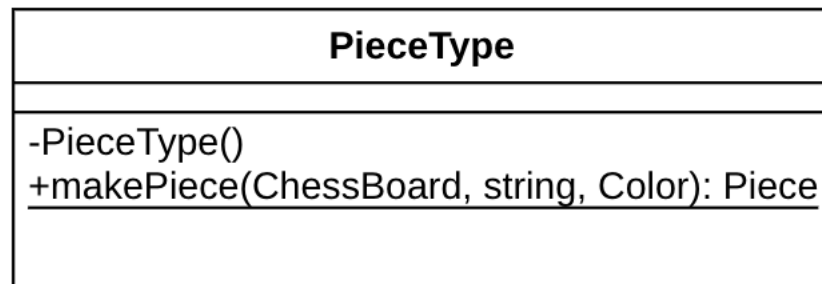
            if(validChecker1.Check(board))
                return true;
        }
    }
    return false;
}
```

3- Design Patterns



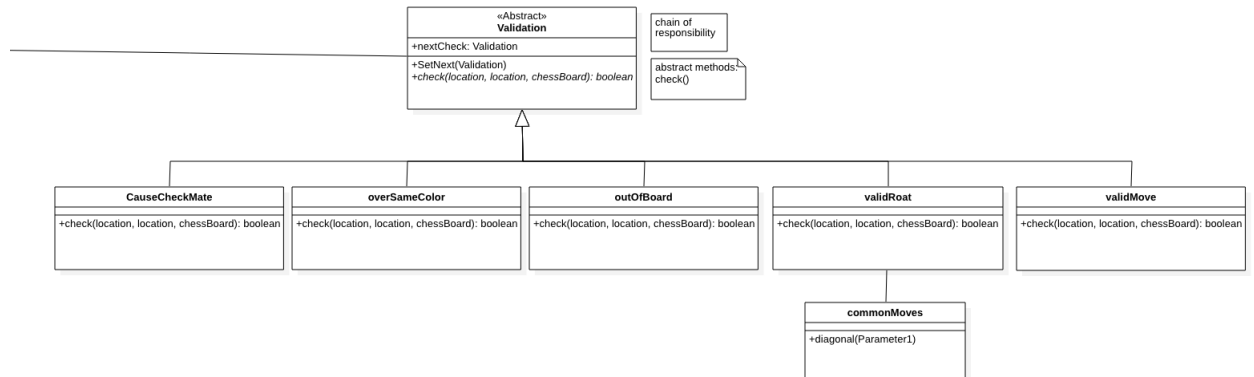
factory method pattern

Initializing the Piece in the square using the **PieceMaker** to make it more readable for the user and avoid using the 'new' operation
initializing different type of chess pieces



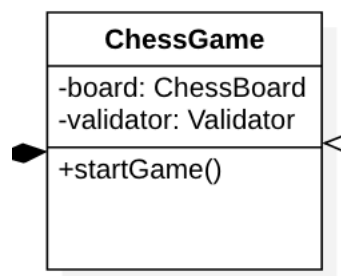
Chain of responsibility patterns

The validation class abstracted into 4 types of validating and use then to check the validity. to make it easier to trace bugs and control the validations need to be checked and skipped and make able to add any new validators in future.



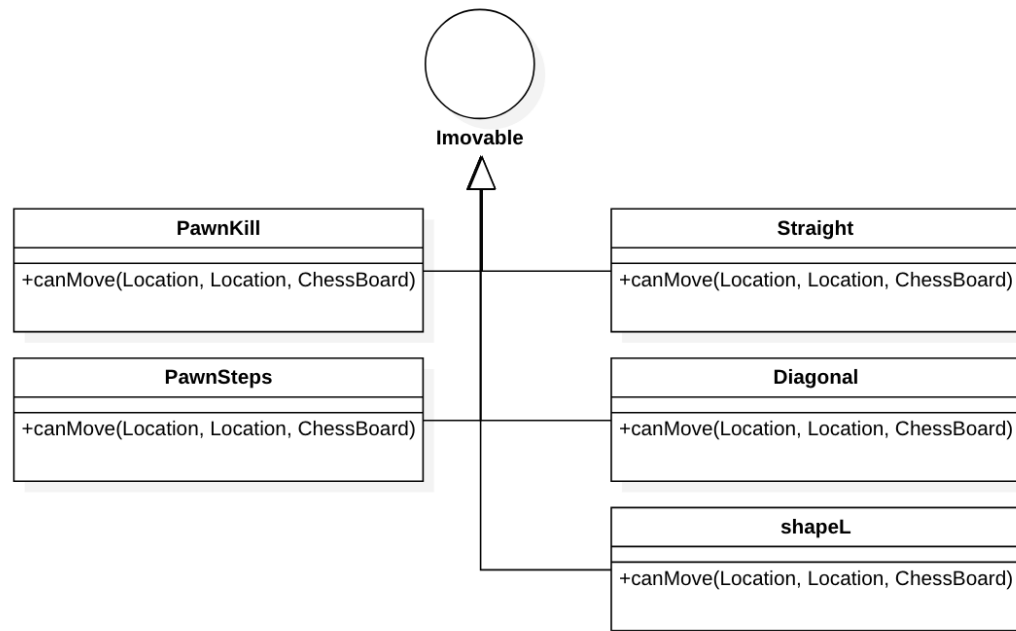
Mediator class patterns

The **ChessGame** class controlling the whole playing process. To keep away from the complicated operations need to be performed.



Strategy design pattern

Some Pieces have a common move behavior such as the Rook and the Queen and the Bishop and Queen so it's better to initialize strategy design pattern to reduce redundancy and make clearer code



4- Solid Principles

Single Responsibility Principle:

the **CommonMoves** class is created to perform the method of moving across the way the piece will move through to check whether there are pieces in the way or not.

The **CheckMate** class is created to check if there is any move that can be performed without causing checkmate, it's used at the beginning of each turn and ends the game when there is no move to perform.

The Dependency Inversion Principle

The **chessGame** mediator depends on the high level module piece and not on the low level modules like king , Knight ...etc.