Report Lab 2 – Verification of pieces using functional decomposition

1. **Requirements**

The program must prompt the user for a chess pieces and its data: type, color, position x and position y. It must store these pieces in its respective object. The program asks the user for a new position and checks if the piece could move to this new position. The program should write “Piece\_Name at X, Y can move to X+1, Y” or “Piece\_Name at X, Y can NOT move to X+1, Y”.

1. **Constraints**

Only one class for each type of piece  
Cannot use inheritance.

The places for the x position are A-H and for the y position 1-8 because a chess board is 8 by 8. There are six different pieces: Pawn, Rook, Knight, Bishop, Queen, and King., with its own movement rules.

The movement rules for each chess piece follows:

* King can move exactly one square horizontally, vertically, or diagonally.
* Queen can move any number of vacant squares diagonally, horizontally, or vertically.
* Tower can move any number of vacant squares vertically or horizontally.
* Bishop can move any number of vacant squares in any diagonal direction.
* Knight can move one square along any rank or file and then at an angle. The knight’s movement can also be viewed as an “L” or “7″ laid out at any horizontal or vertical angle.
* Pawns can move forward one square, if that square is unoccupied. If it has not yet moved, the pawn has the option of moving two squares forward provided both squares in front of the pawn are unoccupied. A pawn cannot move backward.

1. **High-level approach (algorithm)**
2. Prompt user for piece information
3. Create a piece object with the information on the line (corresponding to type, color, and position x and y)
4. Prompt the user for a new position to move to
5. Check if it is a valid position in the chess board, not outside boundaries
6. Call the method to check if the move is valid from the piece object
7. Print if the piece could move to the new position for each piece in the array
8. Ask the user if he wants to test another piece or exit, if he does, return to step 1
9. **Pseudo code for each major function**
10. **List the major parts of your solution (class or methods), show them in a diagram, show their relations**
11. **MAIN LAB CLASS**
    1. **validMoves**
    2. **main – calls the above methods to execute the program**
12. **PAWN CLASS, BISHOP CLASS, KNIGHT CLASS, QUEEN CLASS, ROOK CLASS, KING CLASS**
    1. **Attributes**
       1. **type**
       2. **color**
       3. **posX**
       4. **posY**
    2. **Constructors**
       1. **Piece default**
       2. **Piece with parameters**
    3. **Getters**
       1. **getType**
       2. **getColor**
       3. **getPosX**
       4. **getPosY**
    4. **toString**
    5. **validate**
    6. **showResult**

**Lab2**

scnr  
piece  
positions  
test

validMoves(Scanner)  
main()

**Pawn**

type  
color  
posX  
posY

Pawn()  
Pawn(String, String, Char, int)  
getType()  
getColor()  
getPosX()  
getPosY()  
toString()  
validate(char, int)  
showResult(char, int)

**Pawn**

type  
color  
posX  
posY

Pawn()  
Pawn(String, String, Char, int)  
getType()  
getColor()  
getPosX()  
getPosY()  
toString()  
validate(char, int)  
showResult(char, int)

**Pawn**

type  
color  
posX  
posY

Pawn()  
Pawn(String, String, Char, int)  
getType()  
getColor()  
getPosX()  
getPosY()  
toString()  
validate(char, int)  
showResult(char, int)

**Pawn**

type  
color  
posX  
posY

Pawn()  
Pawn(String, String, Char, int)  
getType()  
getColor()  
getPosX()  
getPosY()  
toString()  
validate(char, int)  
showResult(char, int)

**Pawn**

type  
color  
posX  
posY

Pawn()  
Pawn(String, String, Char, int)  
getType()  
getColor()  
getPosX()  
getPosY()  
toString()  
validate(char, int)  
showResult(char, int)

**Pawn**

type  
color  
posX  
posY

Pawn()  
Pawn(String, String, Char, int)  
getType()  
getColor()  
getPosX()  
getPosY()  
toString()  
validate(char, int)  
showResult(char, int)

