ATYPON Java and DevOps Bootcamp

Chess Game

Nezar Al-Salahat

Instructors:

Fahed Jubair

Motasim Aldiab

Content

1 Introduction

This report is an overview of the issue raised in the chess assignment for Atypon's training. The focus of this report on OOP, Design Patterns, Clean Code, and SOLID.

2 The Problem

2.1 Description

In this assignment, you are required to show your design and implementation of the classes that describe the implementation for a console-based Chess game**.**

When running the game, it should output meaningful messages to interact with the players. For example, here is one possible game run. Note that text in italic is the text entered by the players

####

Enter the white player name: *Fahed*

Enter the black player name: *Ahmad*

Enter next move (white player): *move d2 d4*

Enter next move (black player): *move g8 f6*

2 The Problem

2.2 Important Notes

1. Note that if a player inputs an illegal move, then the game should ask the player to "try again". In other words, a player's turn ends only if a legal move is made.

2. The game ends when either player wins, but a draw is also possible.

3. Special moves such as castling and promotion should be considered.

4. Think about modularity in your design. For example, how can your code be extended if we add new game rules?

5. Think about cohesion and coupling in your classes and methods.

6. Do not start coding immediately. Sit for a second and think first about what classes you should have in your code and what relationships they should have.

2.3 Basis and anomalies

Basis: is the basic moves of pieces on board like :

Knight: L moves

Bishop: Diagonal and Anti-Diagonal moves

Queen: Straight and Diagonal moves

King: 1 step move

Rook: straight moves

Pawn: 1 step (or 2 steps in first)

Anomalies: is a special moves like:

Pawn: diagonal when captured.

Pawn promote

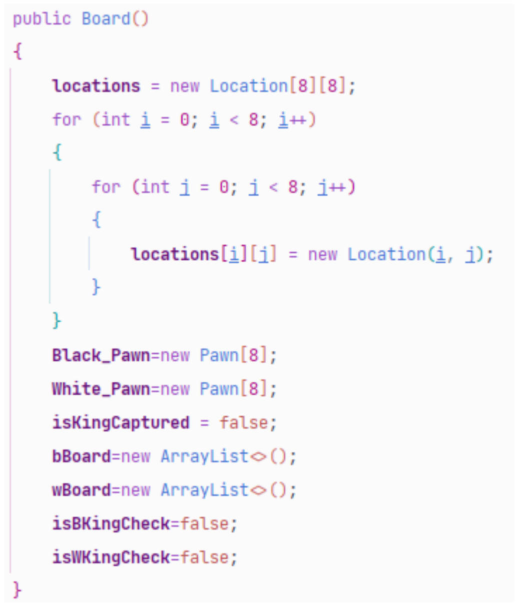
Casting

3 Object-Oriented Design

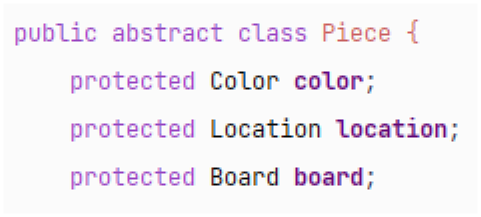
3.1 Description

The code was built in Java and OOP style, so we have classes on it to simplify code and make it more powerful.

The concepts of OOP were covered like; abstraction, polymorphism, encapsulation, composition, aggregation, association and generalization.



```
public Board()
{
    locations = new Location[8][8];
    for (int i = 0; i < 8; i++)
    {
        for (int j = 0; j < 8; j++)
        {
            locations[i][j] = new Location(i, j);
        }
    }
    Black_Pawn=new Pawn[8];
    White_Pawn=new Pawn[8];
    isKingCaptured = false;
    bBoard=new ArrayList<>();
    wBoard=new ArrayList<>();
    isBKingCheck=false;
    isWKingCheck=false;
}
```



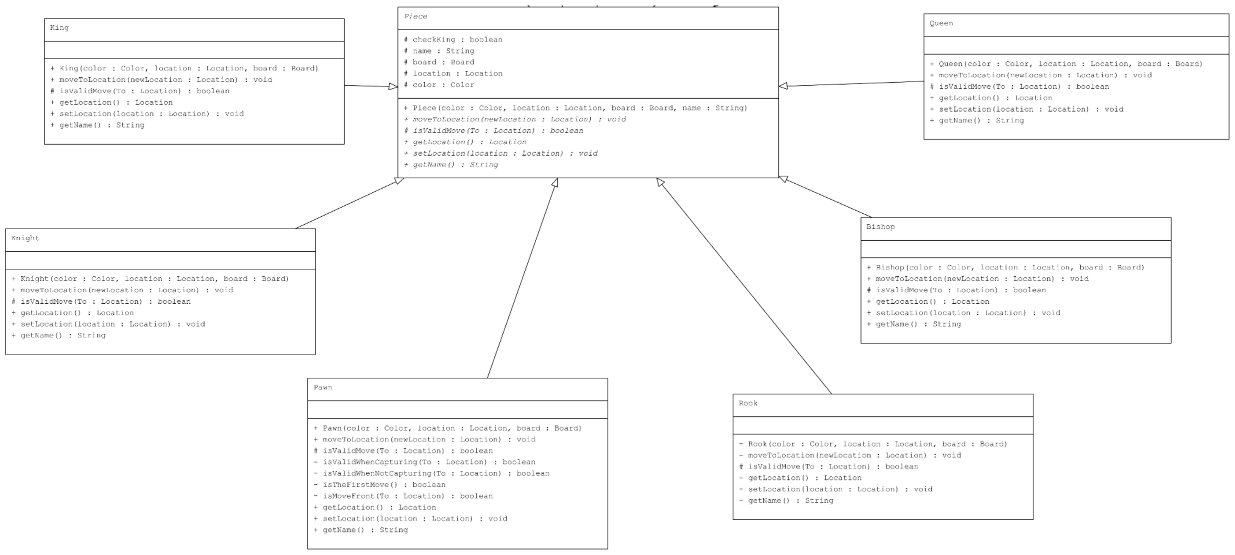
```
public abstract class Piece {
    protected Color color;
    protected Location location;
    protected Board board;
```

The figures show some of the concepts.

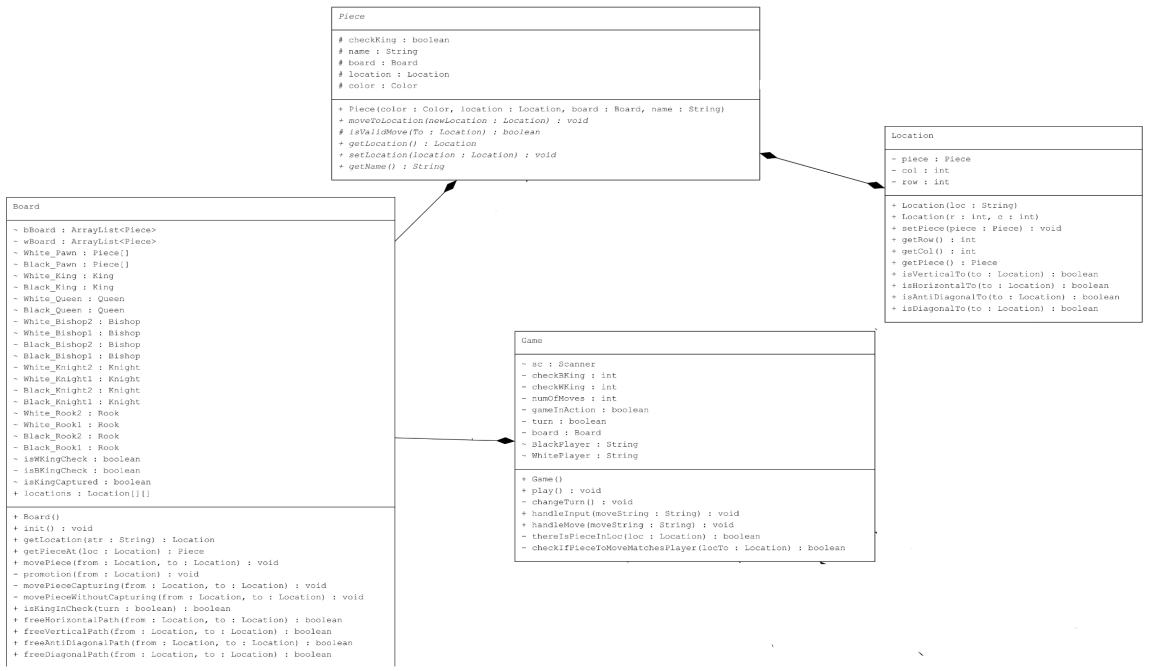
3 Object-Oriented Design

3.2 UML Diagrams

Piece class with pieces:


King
+ King(color : Color, location : Location, board : Board)
+ moveToLocation(newLocation : Location) : void
# isValidMove(To : Location) : boolean
+ getLocation() : Location
+ setLocation(location : Location) : void
+ getName() : String
Piece
# checkKing : boolean
# name : String
# board : Board
# location : Location
# color : Color
+ Piece(color : Color, location : Location, board : Board, name : String)
+ moveToLocation(newLocation : Location) : void
# isValidMove(To : Location) : boolean
+ getLocation() : Location
+ setLocation(location : Location) : void
+ getName() : String
Queen
+ Queen(color : Color, location : Location, board : Board)
+ moveToLocation(newLocation : Location) : void
# isValidMove(To : Location) : boolean
+ getLocation() : Location
+ setLocation(location : Location) : void
+ getName() : String
Knight
+ Knight(color : Color, location : Location, board : Board)
+ moveToLocation(newLocation : Location) : void
# isValidMove(To : Location) : boolean
+ getLocation() : Location
+ setLocation(location : Location) : void
+ getName() : String
Bishop
+ Bishop(color : Color, location : Location, board : Board)
+ moveToLocation(newLocation : Location) : void
# isValidMove(To : Location) : boolean
+ getLocation() : Location
+ setLocation(location : Location) : void
+ getName() : String
Pawn
+ Pawn(color : Color, location : Location, board : Board)
+ moveToLocation(newLocation : Location) : void
# isValidMove(To : Location) : boolean
- isValidWhenCapturing(To : Location) : boolean
- isValidWhenNotCapturing(To : Location) : boolean
- isTheFirstMove() : boolean
- isMoveFront(To : Location) : boolean
+ getLocation() : Location
+ setLocation(location : Location) : void
+ getName() : String
Rook
+ Rook(color : Color, location : Location, board : Board)
+ moveToLocation(newLocation : Location) : void
# isValidMove(To : Location) : boolean
+ getLocation() : Location
+ setLocation(location : Location) : void
+ getName() : String

Piece, Board, Game and Location:


Piece
# checkKing : boolean
# name : String
# board : Board
# location : Location
# color : Color
+ Piece(color : Color, location : Location, board : Board, name : String)
+ moveToLocation(newLocation : Location) : void
# isValidMove(To : Location) : boolean
+ getLocation() : Location
+ setLocation(location : Location) : void
+ getName() : String
Location
- piece : Piece
- col : int
- row : int
+ Location(loc : String)
+ Location(r : int, c : int)
+ setPiece(piece : Piece) : void
+ getRow() : int
+ getCol() : int
+ getPiece() : Piece
+ isVerticalTo(to : Location) : boolean
+ isHorizontalTo(to : Location) : boolean
+ isAntiDiagonalTo(to : Location) : boolean
+ isDiagonalTo(to : Location) : boolean
Board
~ bBoard : ArrayList<Piece>
~ wBoard : ArrayList<Piece>
~ White_Pawn : Piece[]
~ Black_Pawn : Piece[]
~ White_King : King
~ Black_King : King
~ White_Queen : Queen
~ Black_Queen : Queen
~ White_Bishop2 : Bishop
~ White_Bishop1 : Bishop
~ Black_Bishop2 : Bishop
~ Black_Bishop1 : Bishop
~ White_Knight2 : Knight
~ White_Knight1 : Knight
~ Black_Knight2 : Knight
~ Black_Knight1 : Knight
~ White_Rook2 : Rook
~ White_Rook1 : Rook
~ Black_Rook2 : Rook
~ Black_Rook1 : Rook
~ isWKingCheck : boolean
~ isBKingCheck : boolean
~ isKingCaptured : boolean
+ locations : Location[][]
+ Board()
+ init() : void
+ getLocation(str : String) : Location
+ getPieceAt(loc : Location) : Piece
+ movePiece(from : Location, to : Location) : void
- promotion(from : Location) : void
- movePieceCapturing(from : Location, to : Location) : void
- movePieceWithoutCapturing(from : Location, to : Location) : void
+ isKingInCheck(turn : boolean) : boolean
+ freeHorizontalPath(from : Location, to : Location) : boolean
+ freeVerticalPath(from : Location, to : Location) : boolean
+ freeAntiDiagonalPath(from : Location, to : Location) : boolean
+ freeDiagonalPath(from : Location, to : Location) : boolean
Game
~ sc : Scanner
- checkBKing : int
- checkWKing : int
- numOfMoves : int
- gameInAction : boolean
- turn : boolean
- board : Board
~ BlackPlayer : String
~ WhitePlayer : String
+ Game()
+ play() : void
- changeTurn() : void
+ handleInput(moveString : String) : void
+ handleMove(moveString : String) : void
- thereIsPieceInLoc(loc : Location) : boolean
- checkIfPieceToMoveMatchesPlayer(locTo : Location) : boolean

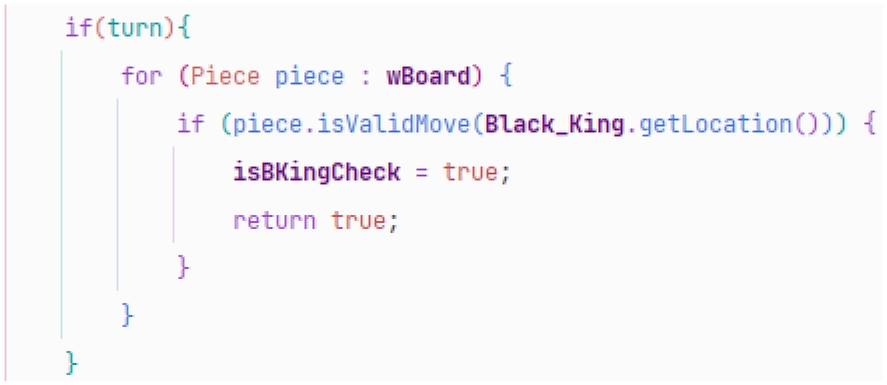
3 Object-Oriented Design

3.3 Functions

I will explain some functions that may make some misunderstandings.

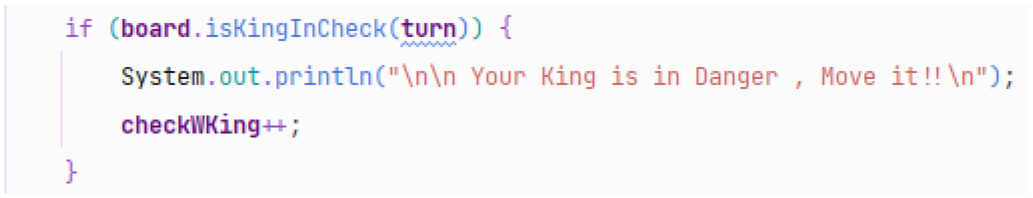
Below function in Board Class, It is used for testing if the pieces on board are checking the king.

wBoard contain all white pieces on board



```java
if(turn){
    for (Piece piece : wBoard) {
        if (piece.isValidMove(Black_King.getLocation())) {
            isBKingCheck = true;
            return true;
        }
    }
}
```

Below code in Game Class, It is used to allow the player that has their king in check to uncheck their king, they have 2 chances then turn changing.



```java
if (board.isKingInCheck(turn)) {
    System.out.println("\n\n Your King is in Danger , Move it!!\n");
    checkWKing++;
}
```

4 Design Pattern

Creational pattern

The game in my code is created on **(Builder Pattern)** because my game generates step by step; firstly Game then Board then Pieces. That provides better control over the construction process and It supports changing the internal representation of objects.

Structural pattern

I didn't use any specific pattern

Behavior pattern

The game in my code is used **(Template Pattern)** because it is used some functions to start the game and we can easily change a game by changing some functions, the code use (init, play and etc..) functions and there are common on all games. I tried to use (Strategy Pattern) in validity but the time was short.

5 Clean Code Principles

5.1 Strength

My code is trying to get all clean code principles.

Naming in Classes, Methods and Variables is clear and match guidelines

In implementation I try to be in the middle between WET and DRY.

In Returning methods, no one returns Null.

Methods take 2 parameters at most.

Using If and else if instead of ternary expressions

Not catch throwable exceptions

My code has a Cohesion

5.2 Weakness

My code hasn’t a SRP (Single Responsibility Principle)

The code has some coupling especially in Validations and Movement.

6 SOLID Principles

6.1 Strength

LSP in promotion

OCP

6.2 Weakness

My code hasn't a SRP (Single Responsibility Principle)

No ISP.

No DIP

7 Conclusion

My code works as a simple Chess game but I have some weaknesses because of promotion and casting.

I hope the code interest you

This is link of YouTube Video: https://youtu.be/6zWCxvBwXeo

Drive Code: https://drive.google.com/drive/folders/1wxQ5K96pJuSlqJvDPl3G1j4Md3LIfF0Y?usp=sharing

Thank you