# Chess Game Assignment

Atypon: Devops Training – 2023 / Jan.

Omar Ramadan – CS / JUST Student

This report to Implement some important concepts of programing:

- OOP
- SOLID principles
- Design patterns
- Clean Code

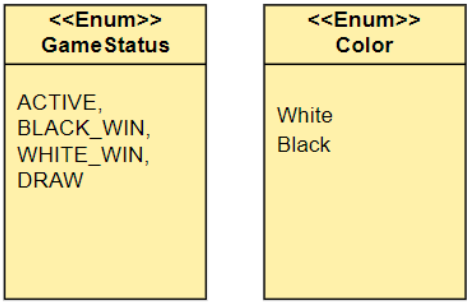
# Introduction:

The task was to write a fully functional Console-based chess game, and to show the implemented design. In this report, I will show my design and discuss it, discuss the used design patterns, defend my code against SOLID and clean code principles.

# My Classes Description and Relation Between them :

## - Enumerations

First of all I created some Enumerations to increase the readability of my code and They allow you to use descriptive names for values and They can help to enforce type safety and prevent the use of invalid values and some of my Enumerations is:



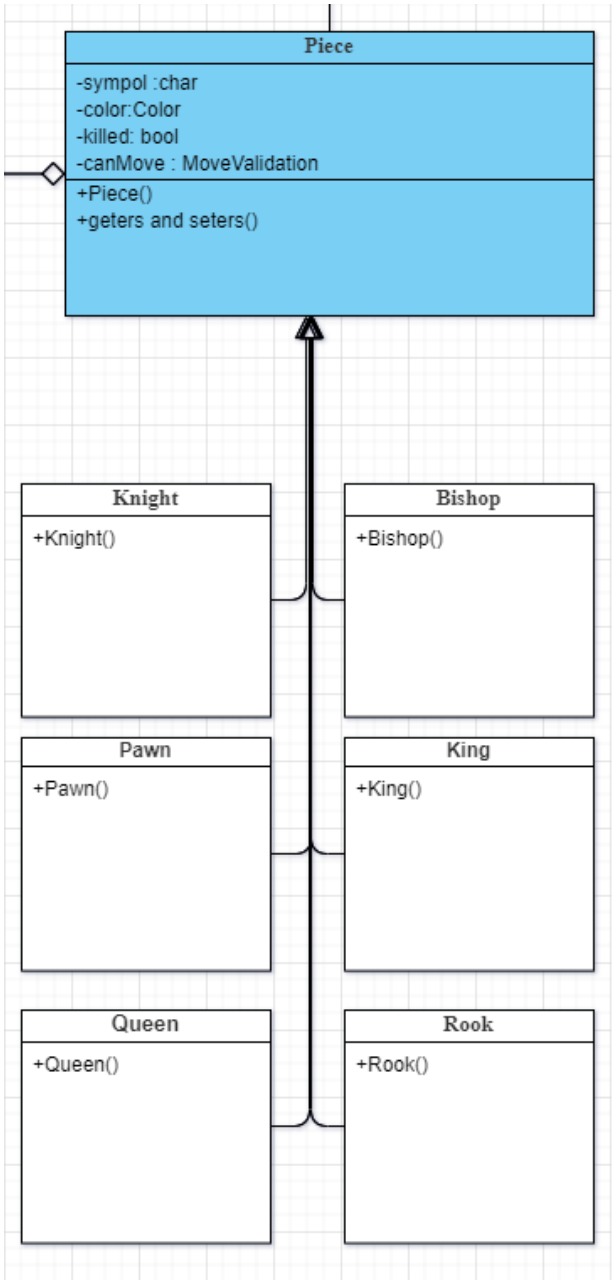
1 – Color Enum : this Enum have two values “Black” and “White” to represent the color of the chess pieces

2 – Game Status: this Enum To Know what is the game status, so if the game was “Active“ will continue, else will stopped and print the proper message that fit the position

- Pieces Package:

This Package responsible about represent the pieces so it has super class named Piece and this class have some main attribute such as color, symbol, canMove and killed this attribute will play the big role in this package and there are few method to like the constructor and some other getters and setters, the constructor will get the color of the piece and set all the value to its defaults

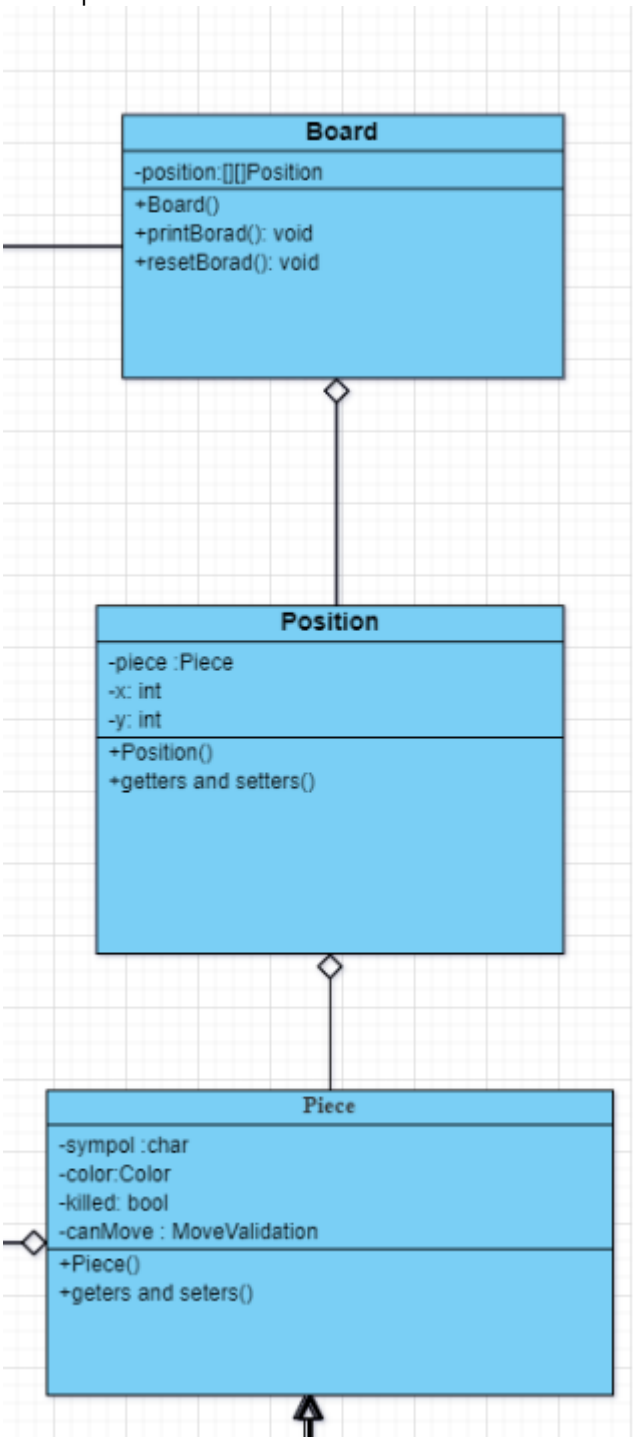
After this there is punch of sub classes(Queen, King, Pawn, Knight, … ) extend the super class (Pieces), these sub class have constructor to set the color value and to make the right move based on the Piece it self.




- Board Class:

These class responsible about setup the board and initiate every position of it by making Array of [8][8] of type Positions,

And make some important things like setup and reset the board and print the values of the board,




- Position class:

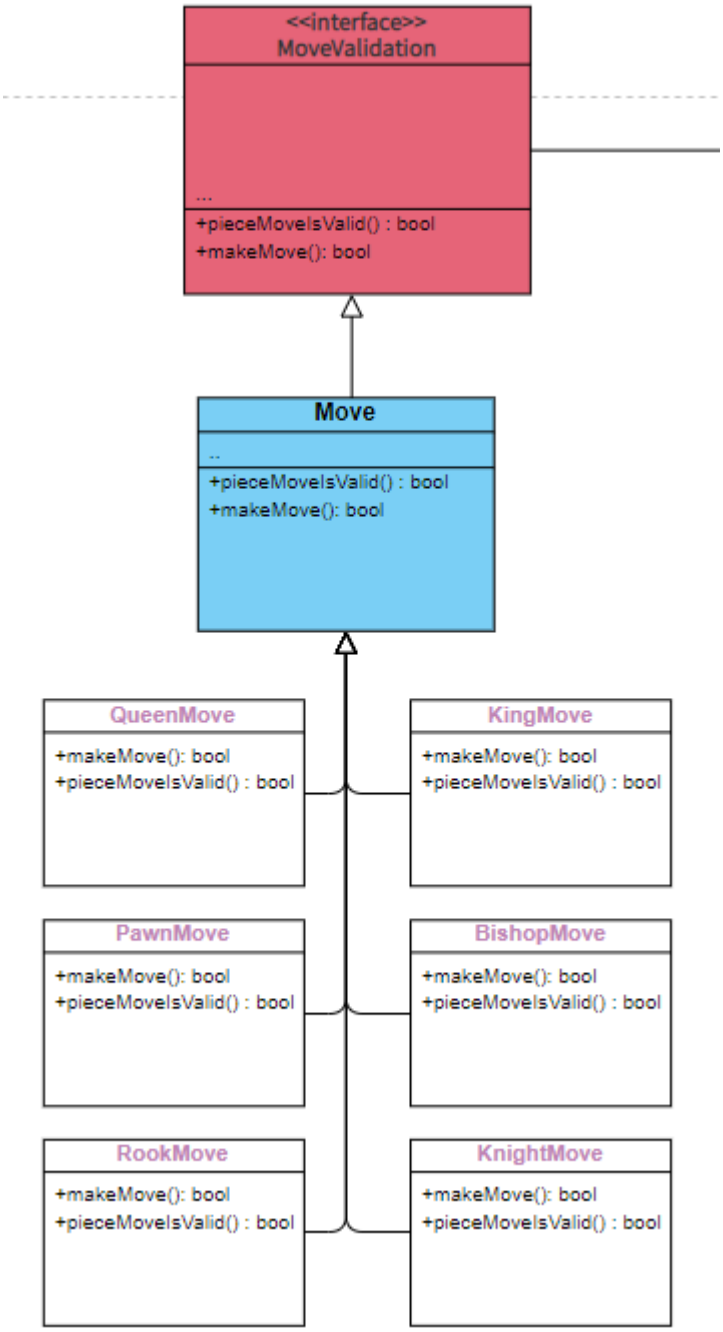
Position is class witch extend the Board, Position class have important attribute like x, y coordinate of the board and it's the super type of Piece class so he will be consist of three things: X, Y and Piece

And he is the percussion officer in this Project,

The relation between them is aggregation because if someone doesn't exist the others will not.

- MoveValidation Interface:

These interface will be super to all the basic movies possible in chess witch will have two method to override witch it pieceMoveIsValid() to check if the piece can make right move and makeMove() to make the move if it is valid




- Move Class:

These class is implements the above class and make some validation on the piece and override the methods above

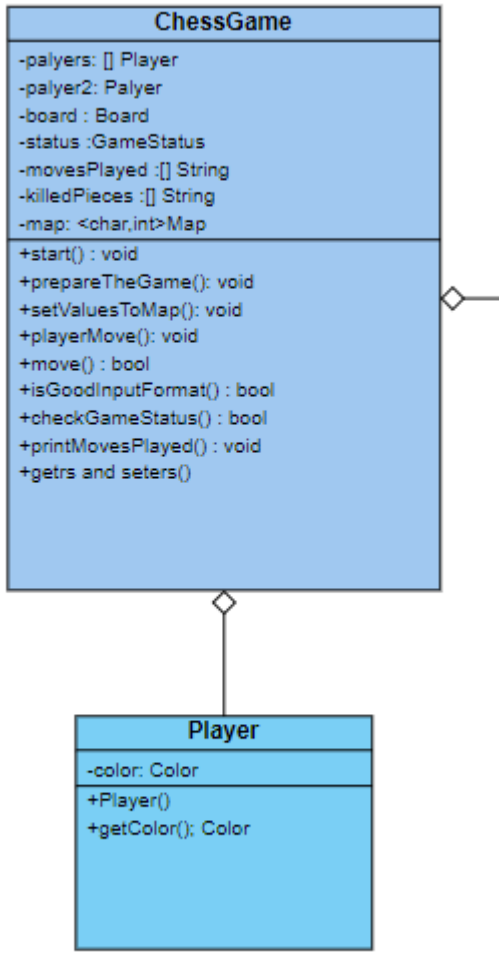
- Sub Class:

This class is extend the Move class and make check how each piece will move

- ChessGame Class(Main):

the main class and the most important of all the project because it will combine all the class together and start the game

these class have important attribute like the players, the bord and the status of the game and will contain some list to track the killed piece and also track the moved played.

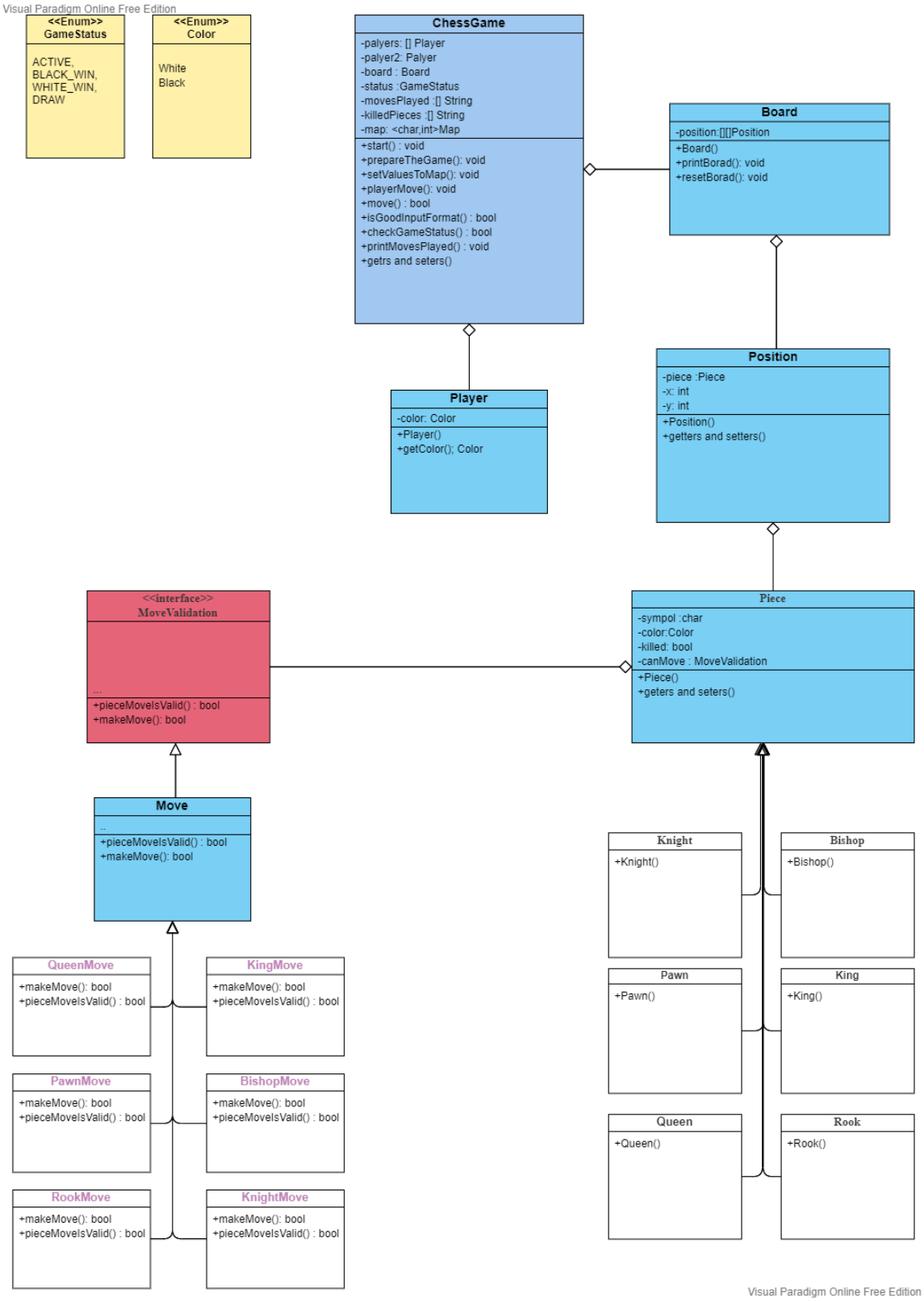



It's also have important method like start() to start the game, preparTheGmae() witch will make setup the board and other things like lists, PlayerMove() to define witch players turn,

isGoodInputFormat() to validate the input format and some getters and setters().

## Player Class :

These class responsible about make difference between users and the relation between the two class is Aggregation because if the chess game not found then the players will not.



# Defending Against SOLID Principles:

## 1- Single Responsibility Principle:

It was made sure that every class and method has only one responsibility, which made the code well organized and made the method's implementations simpler. Although

## 2- Open/Closed Principle:

All classes are stable in the system which implies the closing part of the principle. And it would be easy to add new features to the classes using generalization.

## 3- Liskov's Substitution Principle:

All subtypes only add functionality to super types in the program, although there are no supertypes in the project that are not abstract, but it would cause no problems to replace a supertype with a subtype.

## 4- Interface Segregation Principle:

There is only one interface and no want to splitting it.

## 5- Dependency Inversion Principle:

The idea behind DIP is to decouple the high-level and low-level parts of your system, so that you can change the low-level details without affecting the high-level functionality. This is

achieved by introducing abstractions that define the interface between the high-level and low-level modules.

# Defending Against Clean Code Principles

## 1- Naming Matters / Naming Guidelines:

All names in the project were self-descriptive, you can guess the functionality of a class or a function or a variable just by its name. Also, all naming guidelines for classes, methods, and variables were followed.

## 2- Constructor Chaining:

To follow the DRY principle, in subclasses like those for chess pieces, constructors used the super type constructor to avoid repeating the same implementation.

## 3- Avoiding Returning Nulls:

In some functions in the Board class we return null to represent an empty cell which makes sense, although using the Null Object Pattern would have made sense too, probably I would have used it if I had more time.

## 4- Using Special codes:

Instead of using some hard-to-understand codes, multiple enumerations have been used in the program to keep the code simple, readable, and clean.

## 5- Method Parameters:

Almost no methods accepts more than 3 parameters in the whole project.

## 6- Handling Exceptions:

Exceptions have been handled in all classes.

## 7- Cohesion:

Every class only includes attributes that are strongly connected together. Also, cohesion was taken care of at every package, where every class in every package has some relation to other classes in the same package.

## 8- Coupling:

I tried to reduce coupling between classes as possible, but sometimes it can be so hard to reduce it because of the nature of the project,

make the code understanding easier, this can be clearly seen in the Board and
ValidateMove classes.

## 9- Comments:

No useless comments were used, most of the comments were to indicate the use of a design pattern or to talk about the class functionality in
short. Also, TODOs were used during writing the code to not forget to implement
anything.

# Used Design Patterns

## 1- Facade Pattern:

The ChessGame class used the Facade design pattern, where it collected
the program classes hiding their complexity, and added some logic to them to
make the game run mainly from it. So the ChessGame class basically wrapped the whole project in it.

## 2- The visitor pattern :

was used multiple times in the program to separate the implementations, to keep the classes simple, and to apply the single responsibility principle, so it decoupled some operations from the main classes like the Board and ChessGame classes.

## 3- Singleton Pattern:

I don't use this pattern put maybe if I have more time I can use it with the Scanner and read form the input.

## 4- Factory Pattern:

The factory pattern is a design pattern that provides an interface for creating objects in a super class, but allows subclasses to alter the type of objects that will be created. And I think I use it very will with the piece package

# Testing Process

Testing the project was a bit tricky due to having little bugs so it not the %100 correctness project

I will show you some cases:

### Draw statement :

If the game have 20 move and all the move be like this:
Moves played:
move g1 h3
move g8 h6
move h3 g1
move h6 g8

DRAW

there is some problem with king checkmate so I couldn't do it very will and if you want to win in the game you should eat the king(hhhhh):

### (Fastest)win statement :

move f2 f3
move e7 e6
move e2 e4
move d8 h4

Black Win