**Report**

**Chess Game Assignment**

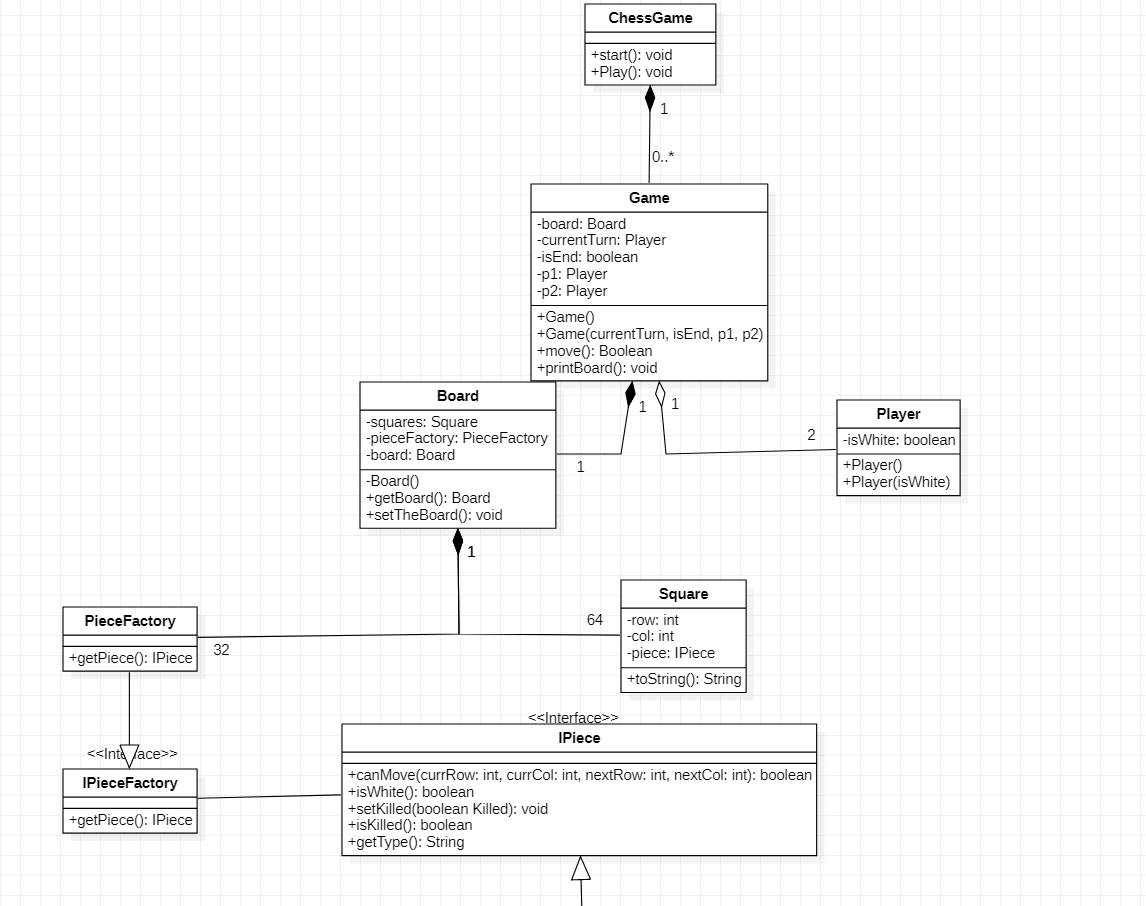
**Name: Diya Momani**

1. Overview :

In this assignment, I focused on demonstrating object-oriented programming thinking by solving the assignment using the object-oriented programming methods. And focus on reducing coupling and increasing cohesion. I also used some design patterns in this assignment to make the code more organized. while following the SOLID principles

1. object-oriented design

My design contains fourteen classes, and this is the UML:



Timeline

Description automatically generated with low confidence

The following is an explanation of each class separately:

* ChessGame Class

This class represents the chess game in general, considering that the chess game will be played many times. Therefore, I decided to separate the number of games that can occur within the chess game.

It consist of one variable:

1-game from Game Type

It consists of 4 functions:

1. Start(): which start the game by invoking play function.
2. Play(): which do the following:

a- initialize and set 2 players

b- set the board

c- while statement that play while game is not end (no king is dead and the number of moves <=50 and there is no checkmate status), and do the following:

* set current player
* players enter values
* check if the entered values are valid
* check if there is check mate status.
* check if the current square has a piece
* check if the player play with piece is not from his pieces
* check if the player will kill one of his pieces
* check if the move for this piece is valid and move them if the move is correct
* print the board

d- print end statement if number of moves are above 50

1. isKingChecked(): check if any pieces Threatens the king

* Game Class

This class represents each Game .And moves the pieces and prints them.

It consists of 5 variables:

1. board from Board Type
2. currentTurn from Player Type
3. isEnd from boolean Type
4. p1 from Player Type
5. p2 from Player Type

It consists of two functions:

1. Move(): which moves the pieces, and do the following:

a-contain boolean variable to check if one of kings dead

b-if statement to check if the next square is null

* if it is null just move the piece
* if it is not null check if the piece is king or not
* if the piece is white king move the black piece to it and kill the king ...then the Black side is win
* if the piece is Black king move the white piece to it and kill the king ...then the white side is win
* if the piece is not king, move the piece and kill it

1. Print(): which print the board

* Player Class

This class represents each player in the game(white and black players).

It consists of one variables:

1. isWhite from Boolean Type

* Board Class

This class represents the game board consisting of pieces and squares

It consist of 3 variables:

1-squares array from Squares Type

2-board from Board Type, and It is a static variable which I will explain it when I explain singleton design pattern

3- pieceFactory from PieceFactory Type, which I will explain it when I explain Factory design pattern

It consists of one functions:

1. SetTheBoard(): set the board and place every piece in its correct square

* Square Class

This class represents the squares on the board, and shows their dimensions and the pieces on them.

It consist of 3 variables:

1. row from int Type
2. col from int Type
3. piece from IPiece Type

It consists of one functions:

1. override toString Function, and if there is piece in this square, print the type of piece in this square and its color else print space

* IPieceFactory interface

This interface consists of one function getPiece interface to get pieces from IPiece interface

* PieceFactory Class

concrete class to override getPiece method from IPieceFactory interface

* IPiece interface

applicant of factory design patterns to reduce coupling, and there is 6 classes implement this interface:

1. Queen
2. King
3. Pawn
4. Bishop
5. Rook
6. Knight

It consists of 5 functions:

1. canMove(): return if the piece can move or not
2. isWhite(): return if the piece is white or not
3. setKilled(): set if the piece is killed or not
4. isKilled(): return if the piece is killed or not
5. getType(): return the type of the piece

* Queen Class

This class represent the queen piece , and consists of two variables:

1. isWhite from boolean Type
2. isKilled from boolean Type

and override all IPiece interface maethods

* King Class

This class represent the king piece , and consists of two variables:

1. isWhite from boolean Type
2. isKilled from boolean Type

and override all IPiece interface maethods

* Bishop Class

This class represent the bishop piece , and consists of two variables:

1. isWhite from boolean Type
2. isKilled from boolean Type

and override all IPiece interface maethods

* Pawn Class

This class represent the pawn piece , and consists of two variables:

1. isWhite from boolean Type
2. isKilled from boolean Type

and override all IPiece interface maethods

* Knight Class

This class represent the knight piece , and consists of two variables:

1. isWhite from boolean Type
2. isKilled from boolean Type

and override all IPiece interface maethods

* Rook

This class represent the rook piece , and consists of two variables:

1. isWhite from boolean Type
2. isKilled from boolean Type

and override all IPiece interface maethods

1. Design Patterns:

I used 2 Design Patterns in this assignment to reduce coupling and organize the code:

1. Singleton design pattern: I used it in Board class by initializing a value of Board type , Thus, I only have one instance board in each game, and I avoid common coupling by using this patterns because I avoid statics and global variables .
2. Factory design pattern: I used it in IPiece interface by associate it with IPieceFactory interface that consists of getPiece() abstract function . PieceFactory class is a concrete class that implements IPieceFactory class and override getPiece() function by passing the piece name and color , and create new object depends on these parameters. IPiece interface consists of 5 functions that are implemented by all pieces types.

By using this pattern I avoid control coupling between classes and make my code more organized.

1. SOLID and clean code(uncle Bob) principles:

I tried to apply the five principles in the code, and below I will explain how:

1. Single responsibility principle: every class in my code do specific thing and solve specific problem ,even the name of classes and functions are clear and you can know what classes are and what functions do from their names, and do what expected to do , thus ,I increase cohesion by applying this principle.
2. Open/ Closed principle: the classes are open for extension, and closed for modifications, in my code IPiece interface applies this principle, so that there is no need to modify the interface in the event of adding a new piece.
3. Interface Segregation principle: I avoid making general interface contains all methods, every interface is specialized with one subject and contains related functions, thus ,I increase cohesion by applying this principle.
4. Dependency Inversion principle: Higher level classes don’t know the implementation of low level classes but depends on abstraction. Below is an example showing the use of this principle in my code:

The Board class does not depend directly on concrete classes. Rather, it deals with the PieceFactory class first (creating an object from it) that deals with the IPieceFactory interface, then the IPieceFactory interface associate with the IPiece interface, and thus I have achieved high dependency in the code.