Chess Game Design

Written by: Moayad AL-SALEH

Table of Contents

What is the problem?

The required to show your design and implementation of the classes that describe the implementation of a Chess game.

Important remarks:

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

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

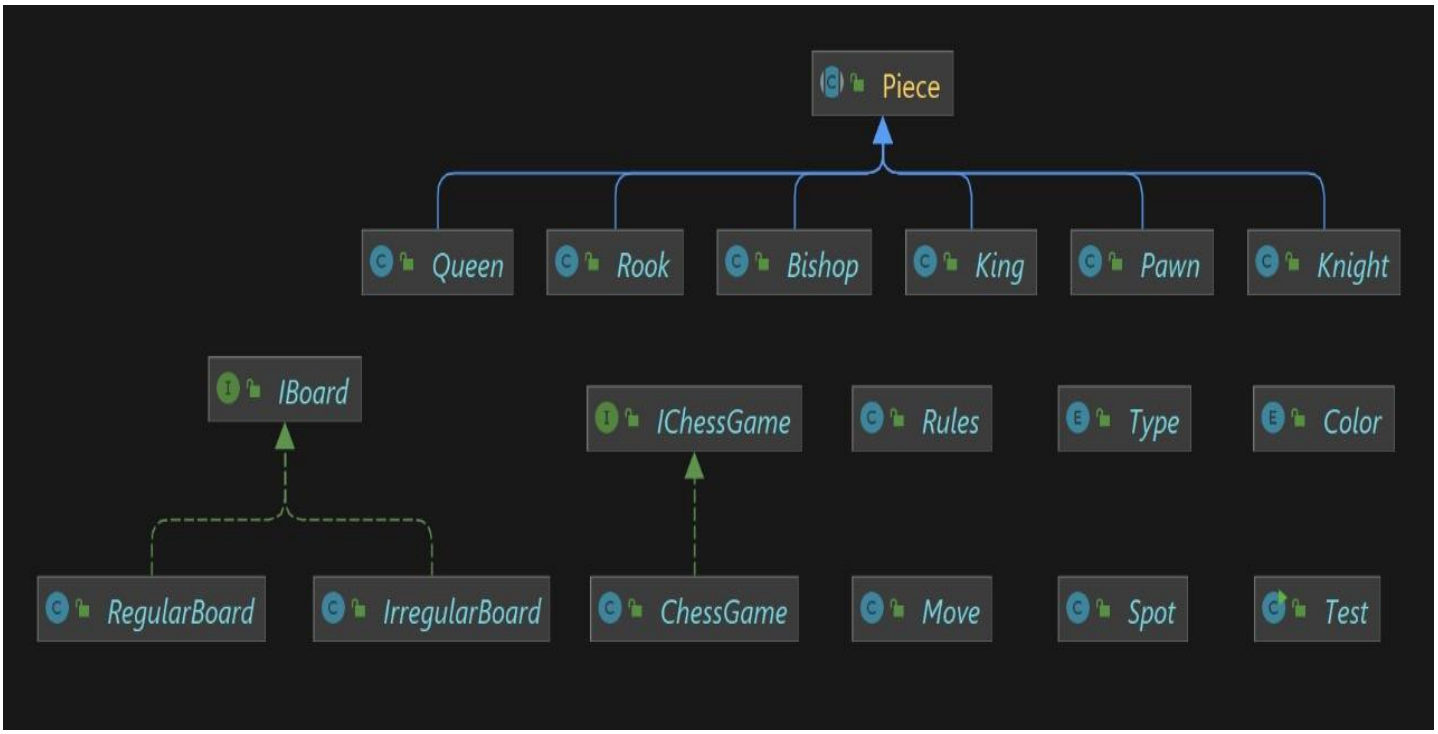
3. Avoid "over-engineering"

4. If the players 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.

5. To reduce coding time, stub-out all methods that are responsible for validation. In other words, show the methods but do not implement them.

6. Finally, you are NOT allowed to make changes on the client code given by me.

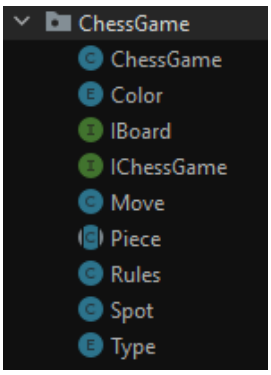
UML Diagram



Object-Oriented Design in code

I applied the principles of the Object-Oriented in the packages.

Chess Game Package:



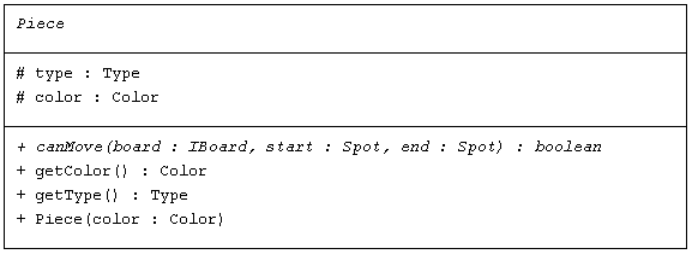
The main components of the package Chess Game

Chess Game Class:

ChessGame
- order : int - formatCorrect : boolean - blackPlayer : String {readOnly} - whitePlayer : String {readOnly} - Winner : String - board : IBoard {readOnly} - kingB : Spot - kingW : Spot - end : Spot - start : Spot
+ ChessGame(W : String, B : String, x : int, y : int) + ChessGame(W : String, B : String) + isDone() : boolean + isWhiteTurn() : boolean + isBlackTurn() : boolean + playWhite(move : String) : void - isCheckmateW() : boolean + playBlack(move : String) : void - isCheckmateB() : boolean - setTheFormat(move : String) : void - isFormatValid(s : String[]) : boolean - isMoveCorrect(color : Color) : boolean - checkWinner(color : Color) : void + printWinnerName() : void

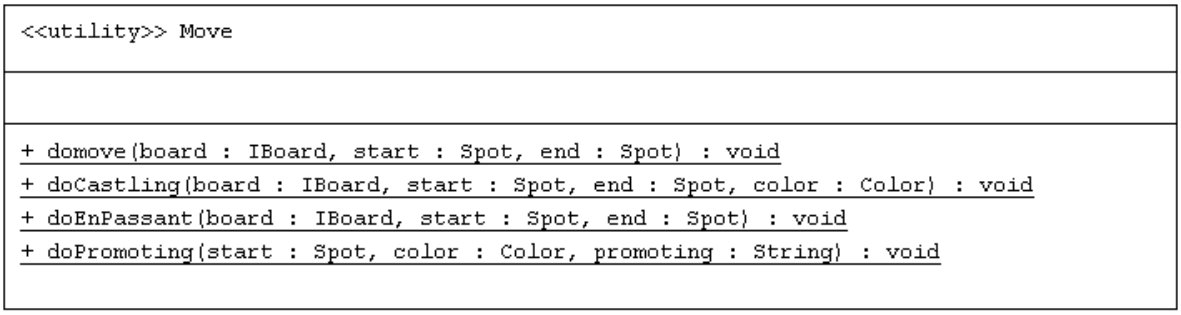
This class contains all the methods in a game such as receiving the movement and sending it for verification and printing the winner of the game.

Piece Class:



I made this class like the father, pieces inherit color and type attributes, and a method to verify the movement of the piece according to the rules.

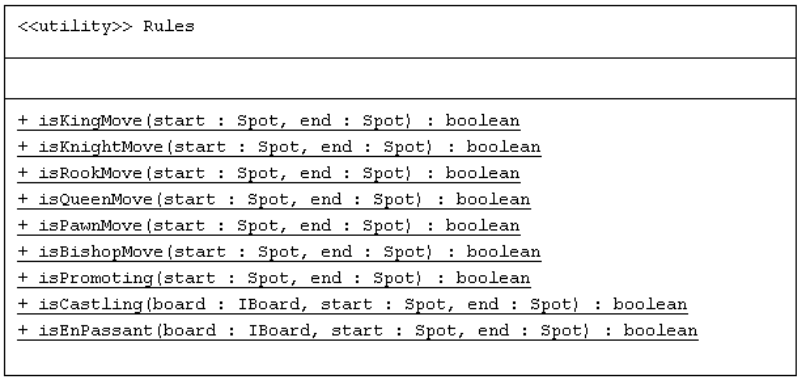
Move Class:



I put in this class all the methods that move the pieces on the chess board.

I designed it so that it allows you to add any movement you want to apply to the pieces, just add the method.

Rules Class:

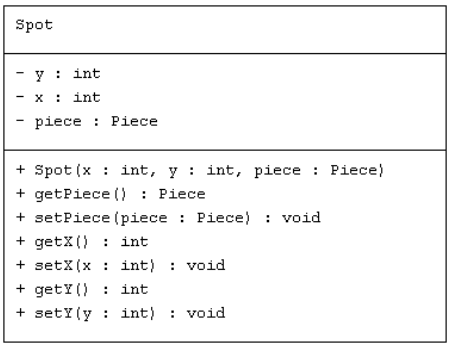


This class contains the rules that you want to apply to the pieces, it is very flexible so it allows the creation or deletion of any rule, this rule is added to the pieces with ease.

```java
@Override
public boolean canMove(IBoard board, Spot start, Spot end)
    {
    return Rules.isQueenMove(start,end) ;// Add any rules you want
    }
```

An example of the Queen piece and how the rule is added to it.

Class Spot:



This class represents the location of each piece separately.

Enum Color, Type:

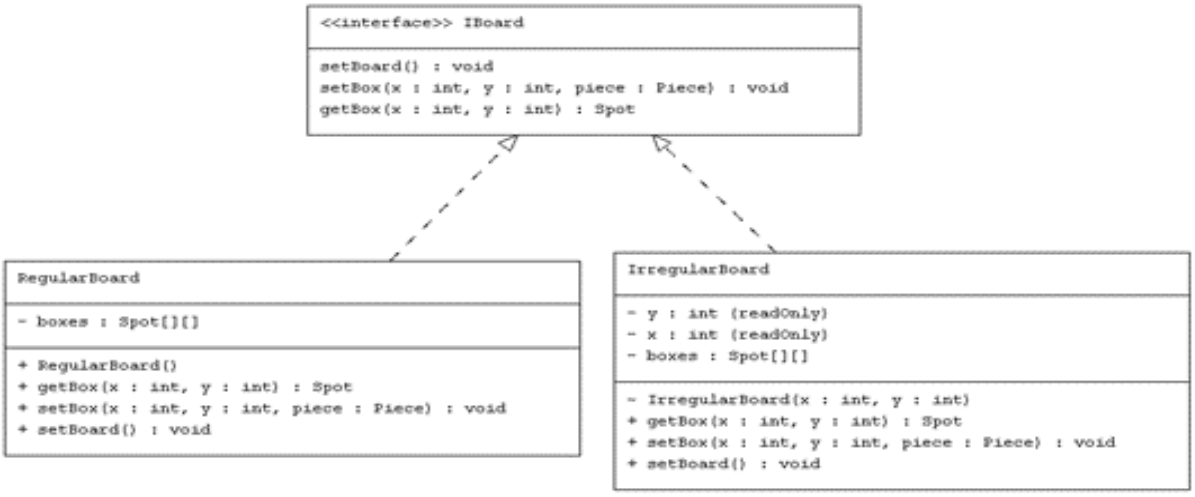
```java
public enum Color
    {
        WHITE, BLACK
    }
```

```java
public enum Type
    {
        ROOK , QUEEN , PAWN ,KNIGHT ,KING ,BISHOP
    }
```

I made **type** and **color** enums so that it is easy to add and delete more **types** and **colors**.

Board Package

Interface Board:



I made the main board as interface, other boards implement its methods.

Piece Package

This package contains all the chess pieces that inherit from Class Piece.

```java
public class Bishop extends Piece
    {
    public Bishop(Color color)
        {
        super(color);
        this.type= Type.BISHOP;
        }

    @Override
    public boolean canMove(IBoard board, Spot start, Spot end)
        {
        return Rules.isBishopMove(start,end);
        }
```

An example of a class in this package is Bishop Class.

Clean Code principles

Code is clean if it can be understood easily – by everyone on the team. Clean code can be read and enhanced by a developer other than its original author. With understandability comes readability, changeability, extensibility and maintainability.

I tried to stick to the principles of **Clean Code** so, I will mention the principles I applied in my code:

General rules:

- Follow standard conventions.
- Keep it simple stupid. Simpler is always better. Reduce complexity as much as possible.
- Boy Scout rule. Leave the campground cleaner than you found it.
- Always find root cause. Always look for the root cause of a problem.

Design rules

- Follow Law of Demeter. A class should know only its direct dependencies.
- Prevent over-configurability

Understandability

- Be consistent. If you do something a certain way, do all similar things in the same way.
- Use explanatory variables.
- Prefer dedicated value objects to primitive type.
- Avoid negative conditionals.

Names rules

- Choose descriptive and unambiguous names.
- Make meaningful distinction.
- Use pronounceable names.
- Use searchable names.

Functions rules

- Small.
- Do one thing.
- Use descriptive names.
- Prefer fewer arguments.

Objects and data structures

- Hide internal structure.
- Prefer data structures.
- Should be small.
- Do one thing.
- Small number of instance variables.

Effective Java

I was not able to heed all the advice in the (**Effective Java**) but what I was able to implement the following:

- ❖ Prefer interfaces to abstract classes
- ❖ Use interfaces only to define types
- ❖ Prefer class hierarchies to tagged classes
- ❖ Minimize the scope of local variable
- ❖ Avoid Strings where other types are more appropriate
- ❖ Avoid unnecessary use of checked exceptions
- ❖ Minimize the accessibility of classes and members
- ❖ Favor composition over inheritance
- ❖ Use enums instead of constants
- ❖ Design method signatures carefully

SOLID principles

SOLID is one of the most popular sets of design principles in object-oriented software development. It's a mnemonic acronym for the following five design principles:

❖ Single Responsibility Principle

(Each class should have one responsibility, one single purpose)

I have applied this principle, so you will find that each package, class and method is responsible for only one thing.

❖ Open / Closed Principle

(Software entities (classes, modules, methods, etc.) should be open for extension, but closed for modification)

I have also worked on applying this principle as much as I can, so you will find that the classes and the methods can be expanded in the future.

❖ Liskov substitution principle

(Any subclass object should be substitutable for the superclass object from which it is derived)

I have been careful while use the principle of inheritance, so you will find that every Super Class does not extend to his child except the method that he needs.

❖ Interface Segregation Principle

(Clients should not be forced to depend upon interfaces that they do not use)

I tried to make all my interfaces responsible for small matters so that the other classes only inherit what they need.

❖ Dependency Inversion Principle

(High level modules should not depend on low level modules; both should depend on abstractions)

I have worked hard to reduce the reliability between the classes and make each class independent of the other, so I increased the use of the Interface and abstraction.