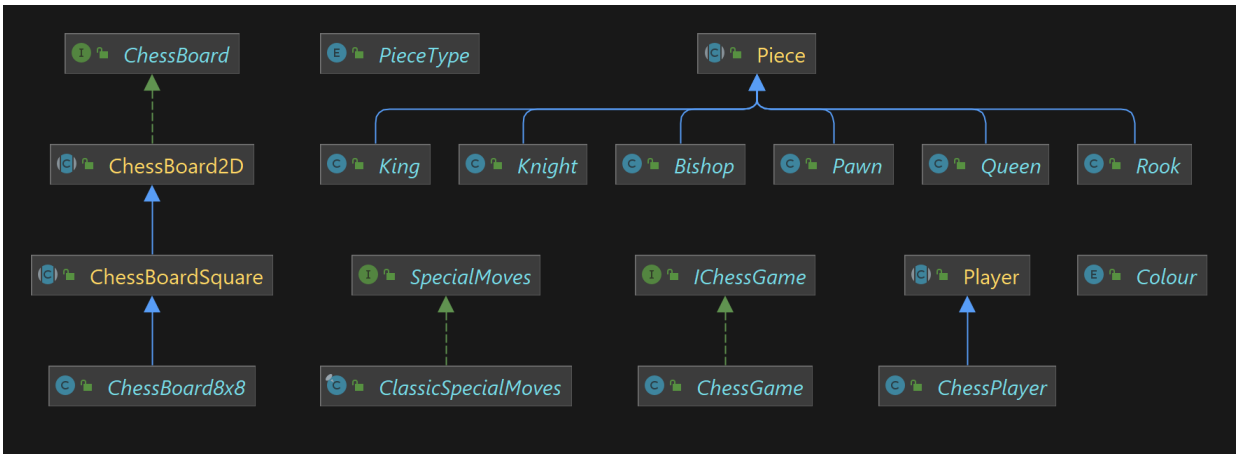
Chess Game Design Report

ESSAM ABDO

Table of Contents

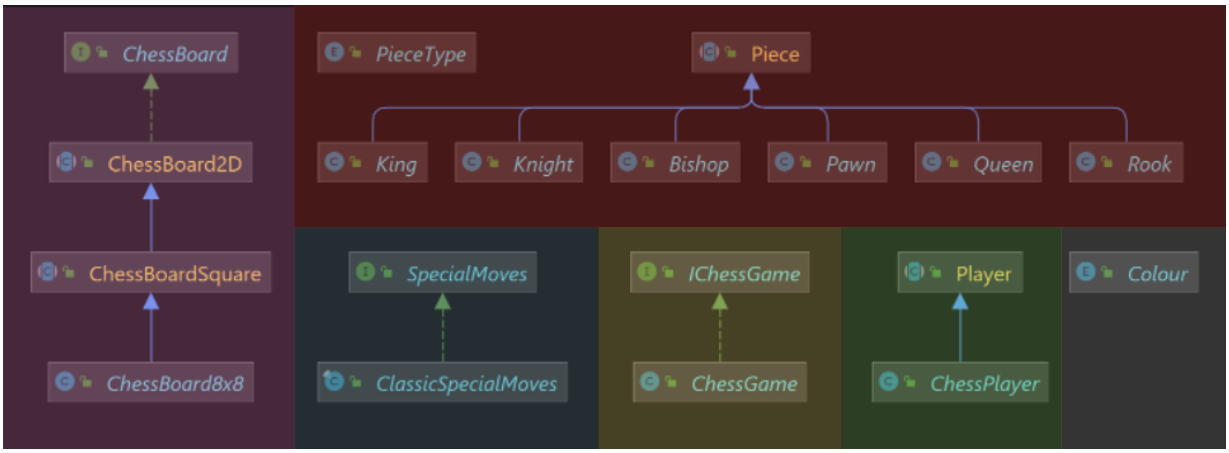
UML Diagram

This is the design I chose to build for Game Chess Assignment :



My design focused mainly on " Separate of concerns principle " , so every class is separated in a different package , and each one of them either implements an interface or extends an abstract class .

This is how each group of classes are organised in packages :



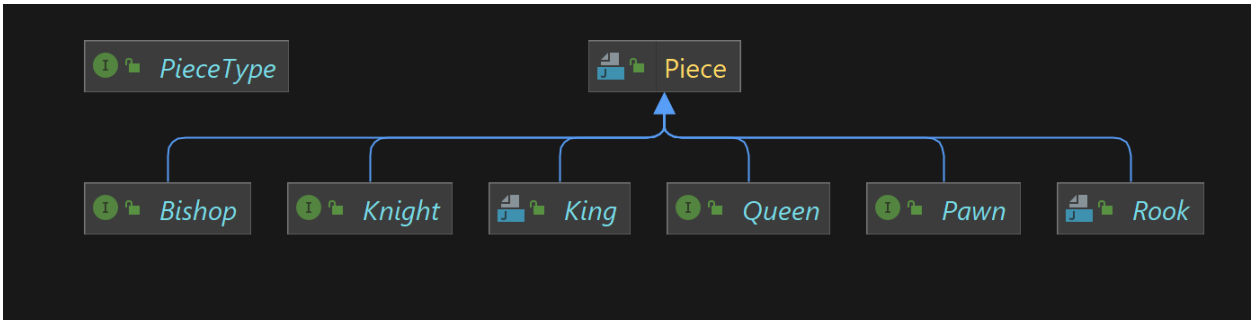
This way , coupling will be sufficiently decreased , and my program will be well cohered (at package level).

Object-Oriented Design

Everything is a class , and each class either extends an abstract class or implements an interface . Colour and PieceType are both enums , for simplicity .

I will describe the structure of each package apart :

pieces package



Piece is an abstract class , and all pieces (king , queen , pawn , etc…) inherits form it .

PieceType is an enum that is frequently used for comparison between the types .

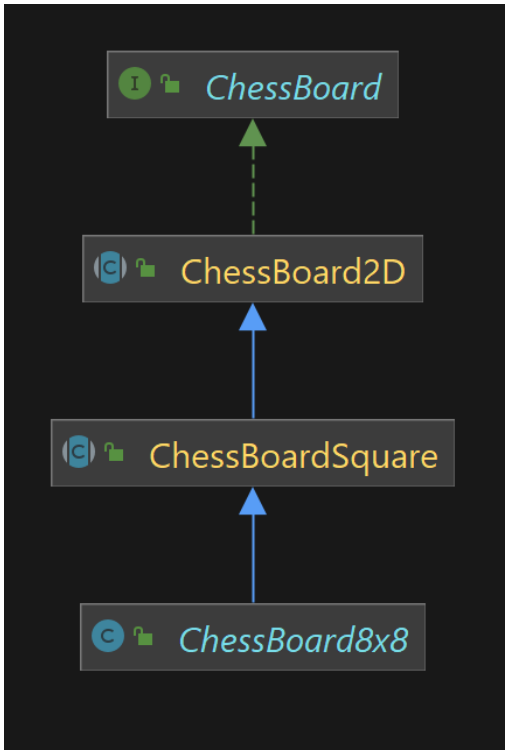
boards package

For boards , ChessBoard is an interface that has all the methods and attributes for a chess board .

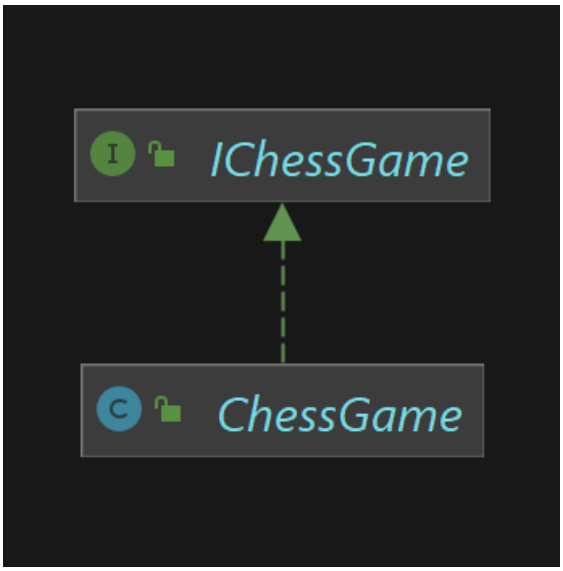
ChessBoard2D is an example of the classes that may implement the interface , if there is a ChessBoard3D , it will be in the same level of this abstract class .

ChessBoardSquare is an abstract class that inherits from ChessBoard2D , in this class the rows and columns of the board are equal .

ChessBoard8x8 is a concrete class that represents a classic chess board , with eight rows and eight columns , and this is the one I used in my chess program .



game package



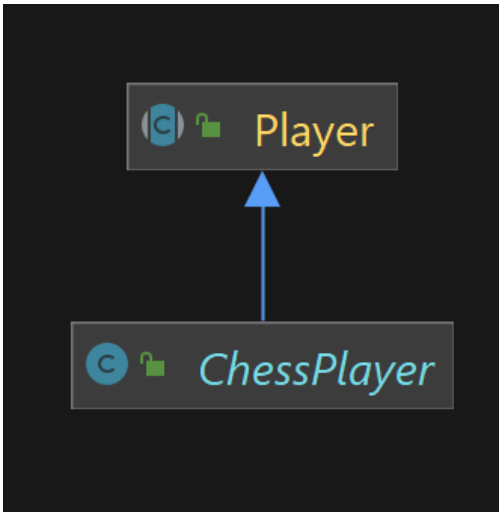
game package consists of IChessGame , which is an interface for the general chess games .

ChessGame is a concrete class that implements IChessGame interface . This is the class that is used for the my chess program .

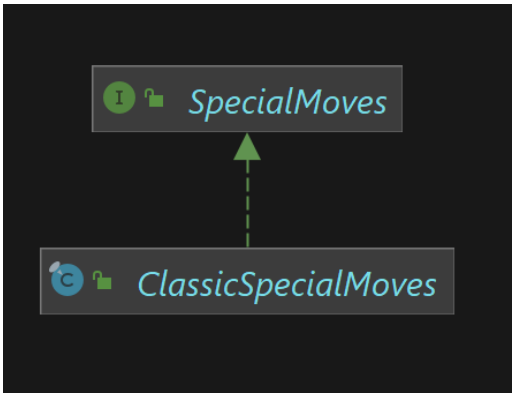
player package

Player package consists of player abstract class , which is like a contract of what attributes a player should have.

ChessPlayer is a concrete class that inherits from player abstract class . This is the class that is used in my chess program .



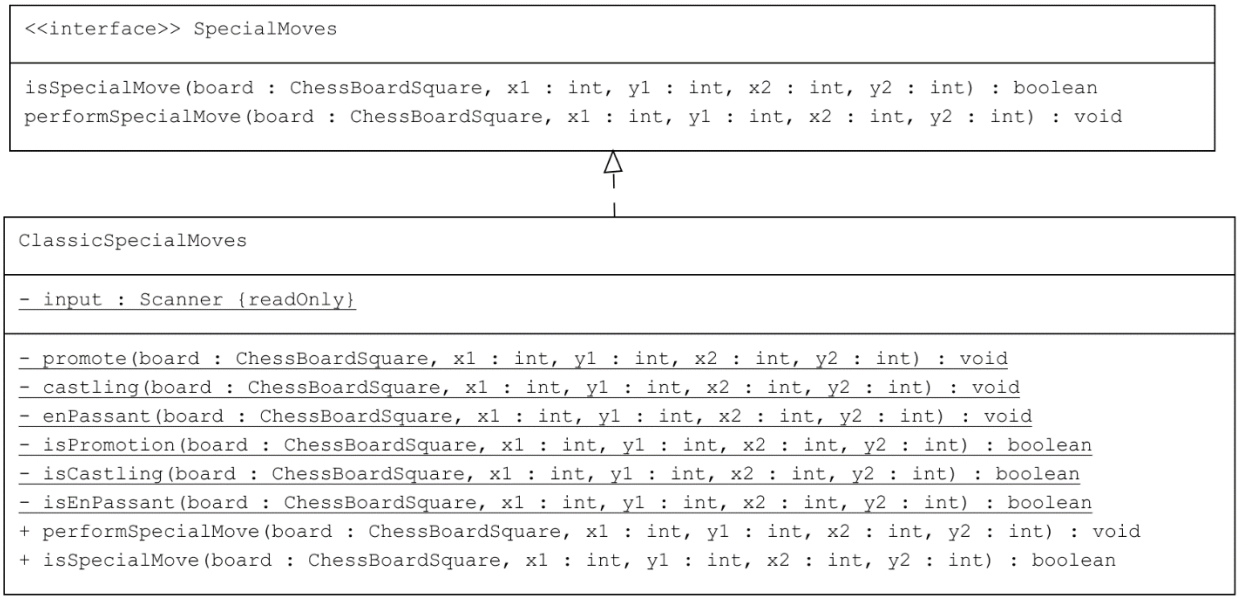
rules package



rules package contains an interface called SpecialMoves , it contains two methods :

- **isSpecialMove()**
- **performSpecialMoves()**

The following is a diagram that shows the methods in this package :



This way , the rules of chess will be very flexible , you can add or remove any rule you like anytime .

colour package

Colour package only has enum that is responsible for colours in the game . By this enum , the operation of adding new colours in the future will not cause a headache .

Clean Code principles

I tried to apply clean code principles as much as I can , here are some of the considerable rules that were applied properly in my code :

General Rules

- I avoided duplication in my code , (DRY principle or Don't Repeat Yourself) .
- I followed a standard convention .
- I was consistent . If I did something a certain way , all similar things are done in the same way .

Names Rules

- I chose descriptive and unambiguous names .
- I chose meaningful names .
- I chose pronounceable names .
- I chose searchable names .
- I made names easy to remember .
- I used names according to the context .
- I used names that are consistent with each other , for example , it's confusing to have "retrieved" and "get" as equivalent methods in distinct classes .

Functions rules

- My functions do one thing .
- I used descriptive names .
- I avoided large functions .

Comments rules

- They were as less as possible .
- I did not add obvious noise .
- I tried to explain myself more in the code .
- I avoided unnecessary comments .

Source Code Structure

- I declared variables close to their usage .
- My similar functions are close to each other .
- I placed the functions in the downward direction .
- I used white spaces to associate related things and disassociate weakly related .

Classes

- My Classes have only one responsibility .
- The base class know nothing about their derivatives .
- I followed Law of Demeter . A class should know only its direct dependencies .

“Effective Java” Items

For “Effective Java” , unfortunately I was not able to read the book , yet , but I have read some summaries of the book , so I can get things done in the current time .

Here are some of the items , that I were able to apply properly on my code :

- I obeyed the general contract when overriding equals .
- I minimized the accessibility of classes and members .
- I used enums instead of int constants .
- Consistently , I used the Override annotation .
- I checked parameters for validity . I checked parameters before execution as soon as possible .
- I designed method signatures carefully .
- I minimized the scope of local variables .
- I avoided Strings where other types are more appropriate

SOLID principles

I have applied and taken consideration of the SOLID principle , while programming the Chess game . For each principle I will demonstrate how this principle was applied in my program :

S : Single Responsibility Principle

" A class should have one and only responsibility over a single part of the functionality provided by the software . "

All classes and methods I wrote , all of them apply this principle , so no method nor class break this principle .

O : Open / Closed Principle (OCP)

" Entities should be open for extension , but closed for modification

(We can apply this using abstract class , and other classes should inherit from this abstract class) . "

All of my classes are either inherits from abstract classes or implements an interface , both of the two options were designed based on the OCP principle .

L : Liskov substitution principle (LSP)

'' Subtypes must be substitutable for their base types

Derived types must be completely substitutable for their base types ''

This principle was applied for in every package .

I : Interface Segregation Principle (ISP)

‘’ One fat interface need to be split to many smaller and relevant interfaces ‘’

This was demonstrated properly in boards package .

D : Dependency Inversion Principle (DIP)

‘’ Depend upon abstraction (interfaces) not upon concrete classes

Abstractions should not depend on the detailed whereas the details should depend on abstractions

High-level modules should not depend on low level modules ’’

Unfortunately , I don’t think that this principle was applied in my code , I feel that I need more observations about this principle so I can know how can I apply it properly in any project .