Chess Game

Assignment 3

Mammon Ahmad Alshamali

18/4/2022

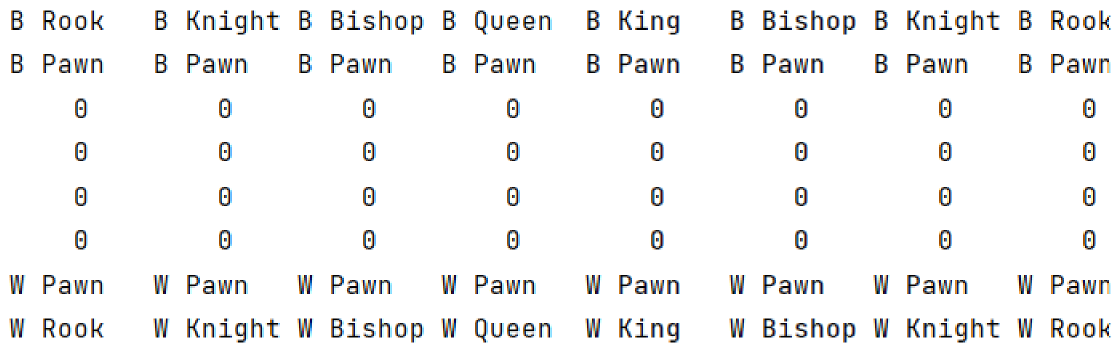
Outline

Introduction

The assignment is about designing and implementing a chess game, there will be some methods that will be stub-out because the main purpose of the assignment is to design and follow Object-Oriented.

The solution will be to defend against The Clean Code principles, "Effective Java" items, and SOLID principles. Also, keeping in mind coupling and cohesion in the classes and methods.

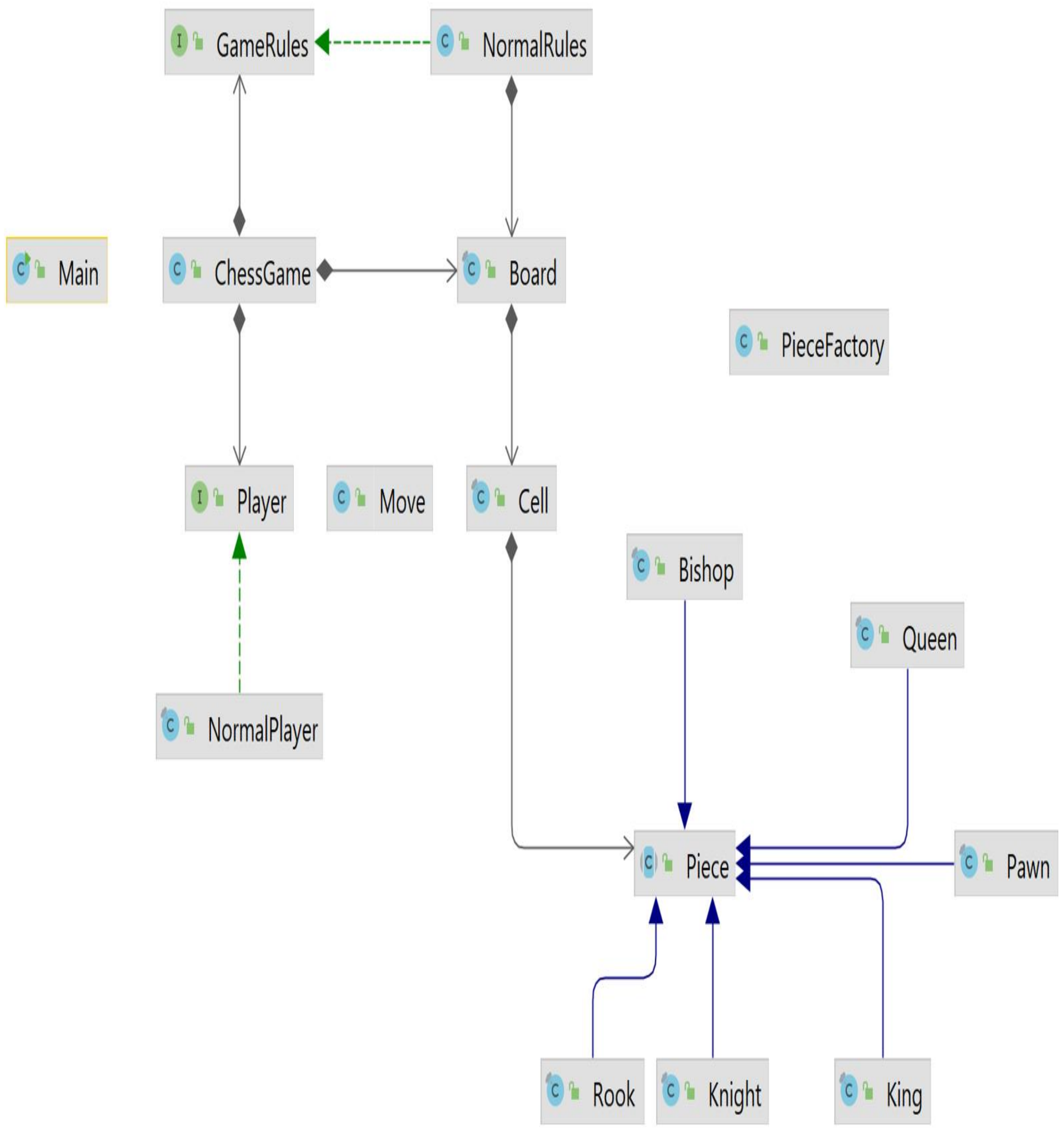
The board will appear in the terminal like this:



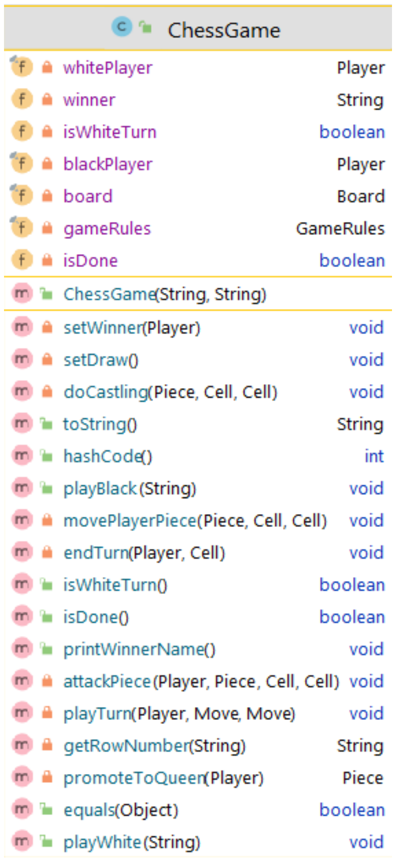
```
B Rook   B Knight B Bishop B Queen  B King   B Bishop B Knight B Rook
B Pawn   B Pawn   B Pawn   B Pawn   B Pawn   B Pawn   B Pawn   B Pawn
    0        0        0        0        0        0        0        0
    0        0        0        0        0        0        0        0
    0        0        0        0        0        0        0        0
    0        0        0        0        0        0        0        0
W Pawn   W Pawn   W Pawn   W Pawn   W Pawn   W Pawn   W Pawn   W Pawn
W Rook   W Knight W Bishop W Queen  W King   W Bishop W Knight W Rook
```

Object-Oriented

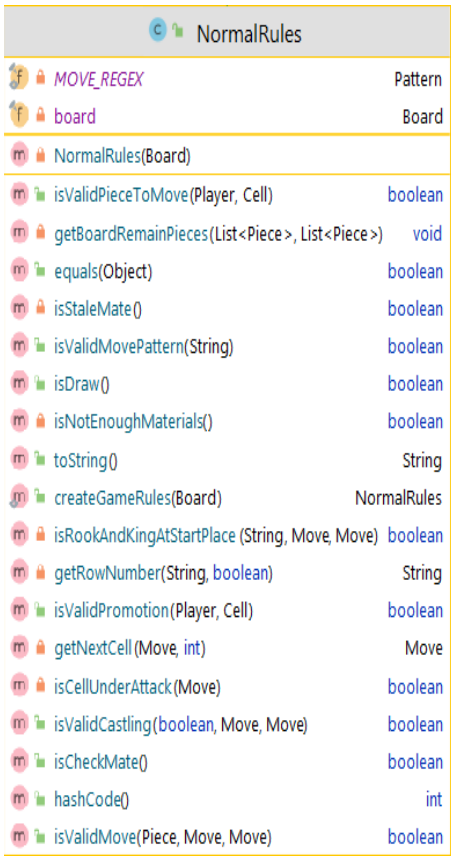
The UML diagram for the chess



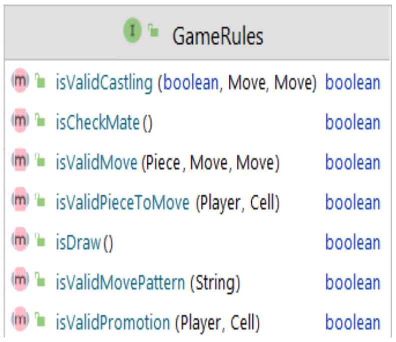
This figure was generated using IntelliJ IDEA Ultimate. Next, the details for each class:



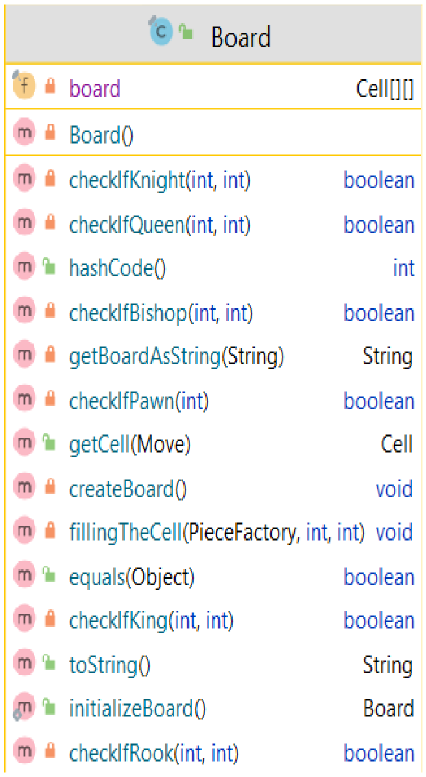
ChessGame	
whitePlayer	Player
winner	String
isWhiteTurn	boolean
blackPlayer	Player
board	Board
gameRules	GameRules
isDone	boolean
ChessGame(String, String)	
setWinner(Player)	void
setDraw()	void
doCastling(Piece, Cell, Cell)	void
toString()	String
hashCode()	int
playBlack(String)	void
movePlayerPiece(Piece, Cell, Cell)	void
endTurn(Player, Cell)	void
isWhiteTurn()	boolean
isDone()	boolean
printWinnerName()	void
attackPiece(Player, Piece, Cell, Cell)	void
playTurn(Player, Move, Move)	void
getRowNumber(String)	String
promoteToQueen(Player)	Piece
equals(Object)	boolean
playWhite(String)	void



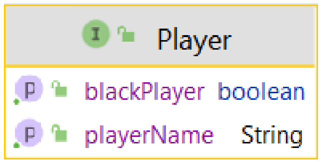
NormalRules	
MOVE_REGEX	Pattern
board	Board
NormalRules(Board)	
isValidPieceToMove(Player, Cell)	boolean
getBoardRemainPieces(List<Piece>, List<Piece>)	void
equals(Object)	boolean
isStaleMate()	boolean
isValidMovePattern(String)	boolean
isDraw()	boolean
isNotEnoughMaterials()	boolean
toString()	String
createGameRules(Board)	NormalRules
isRookAndKingAtStartPlace(String, Move, Move)	boolean
getRowNumber(String, boolean)	String
isValidPromotion(Player, Cell)	boolean
getNextCell(Move, int)	Move
isCellUnderAttack(Move)	boolean
isValidCastling(boolean, Move, Move)	boolean
isCheckMate()	boolean
hashCode()	int
isValidMove(Piece, Move, Move)	boolean



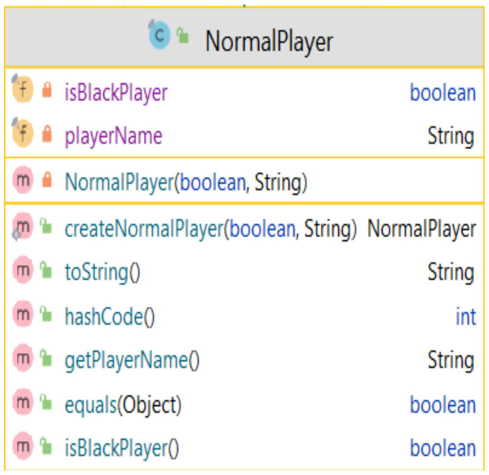
GameRules	
isValidCastling(boolean, Move, Move)	boolean
isCheckMate()	boolean
isValidMove(Piece, Move, Move)	boolean
isValidPieceToMove(Player, Cell)	boolean
isDraw()	boolean
isValidMovePattern(String)	boolean
isValidPromotion(Player, Cell)	boolean



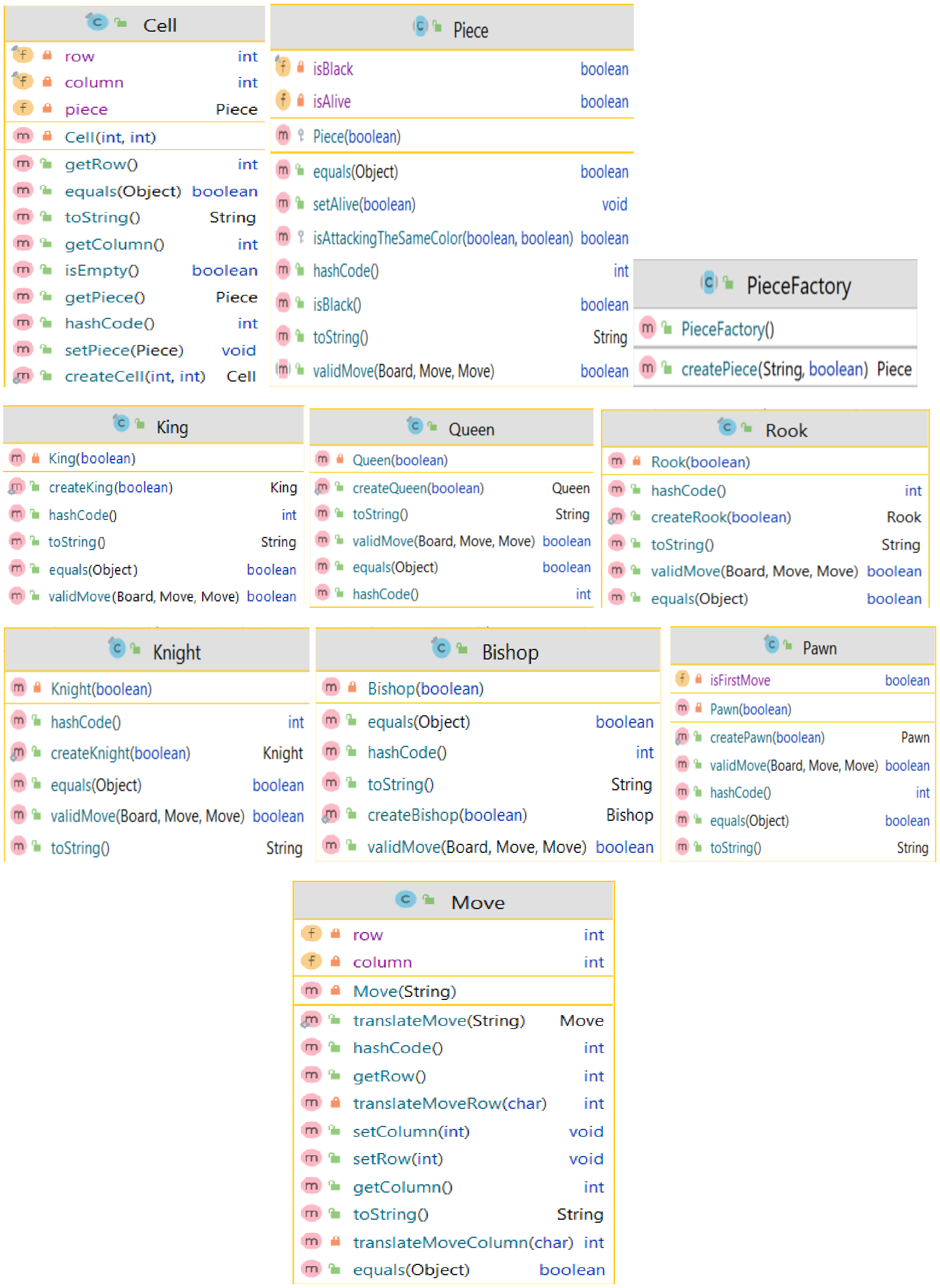
Board	
board	Cell[][]
Board()	
checkIfKnight(int, int)	boolean
checkIfQueen(int, int)	boolean
hashCode()	int
checkIfBishop(int, int)	boolean
getBoardAsString(String)	String
checkIfPawn(int)	boolean
getCell(Move)	Cell
createBoard()	void
fillingTheCell(PieceFactory, int, int)	void
equals(Object)	boolean
checkIfKing(int, int)	boolean
toString()	String
initializeBoard()	Board
checkIfRook(int, int)	boolean



Player	
blackPlayer	boolean
playerName	String



NormalPlayer	
isBlackPlayer	boolean
playerName	String
NormalPlayer(boolean, String)	
createNormalPlayer(boolean, String)	NormalPlayer
toString()	String
hashCode()	int
getPlayerName()	String
equals(Object)	boolean
isBlackPlayer()	boolean



Cell

row	int
column	int
piece	Piece
Cell(int, int)	
getRow()	int
equals(Object)	boolean
toString()	String
getColumn()	int
isEmpty()	boolean
getPiece()	Piece
hashCode()	int
setPiece(Piece)	void
createCell(int, int)	Cell

Piece

isBlack	boolean
isAlive	boolean
Piece(boolean)	
equals(Object)	boolean
setAlive(boolean)	void
isAttackingTheSameColor(boolean, boolean)	boolean
hashCode()	int
isBlack()	boolean
toString()	String
validMove(Board, Move, Move)	boolean

PieceFactory

PieceFactory()	
createPiece(String, boolean)	Piece

King

King(boolean)	
createKing(boolean)	King
hashCode()	int
toString()	String
equals(Object)	boolean
validMove(Board, Move, Move)	boolean

Queen

Queen(boolean)	
createQueen(boolean)	Queen
toString()	String
validMove(Board, Move, Move)	boolean
equals(Object)	boolean
hashCode()	int

Rook

Rook(boolean)	
hashCode()	int
createRook(boolean)	Rook
toString()	String
validMove(Board, Move, Move)	boolean
equals(Object)	boolean

Knight

Knight(boolean)	
hashCode()	int
createKnight(boolean)	Knight
equals(Object)	boolean
validMove(Board, Move, Move)	boolean
toString()	String

Bishop

Bishop(boolean)	
equals(Object)	boolean
hashCode()	int
toString()	String
createBishop(boolean)	Bishop
validMove(Board, Move, Move)	boolean

Pawn

isFirstMove	boolean
Pawn(boolean)	
createPawn(boolean)	Pawn
validMove(Board, Move, Move)	boolean
hashCode()	int
equals(Object)	boolean
toString()	String

Move

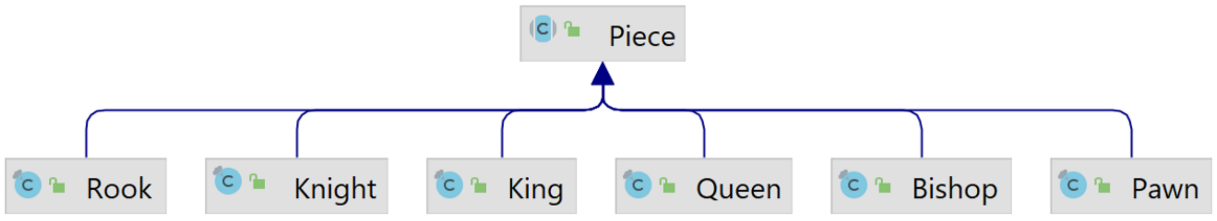
row	int
column	int
Move(String)	
translateMove(String)	Move
hashCode()	int
getRow()	int
translateMoveRow(char)	int
setColumn(int)	void
setRow(int)	void
getColumn()	int
toString()	String
translateMoveColumn(char)	int
equals(Object)	boolean

The UML shows the abstraction and encapsulation of the design. all the classes' properties are private and can be accessed using get and set functions only. What is more, all the functions are private unless are needed for the user to use or it is needed to be used by other objects. As an example, in the chess game class, the promotion and castling are private and the user can only use play white, play black, is done, and print the winner's name. and the rules of the game like a valid promotion or valid castling are achieved using game rules class.

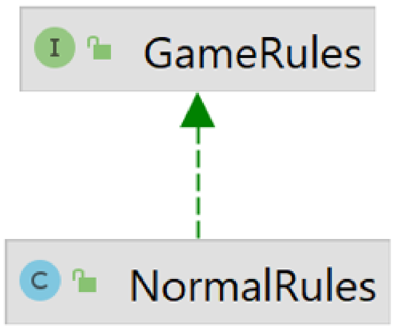
For Decomposition in the design, the following table will explain the relations:

First-class	Second-class	Relation
Chess game	Player	Aggregation
Chess game	Board	Composition
Chess game	Game rules	Composition
Game rules	Board	Aggregation
Board	Cell	Composition
Cell	Piece	Aggregation
Move	Any class use Move class	Association

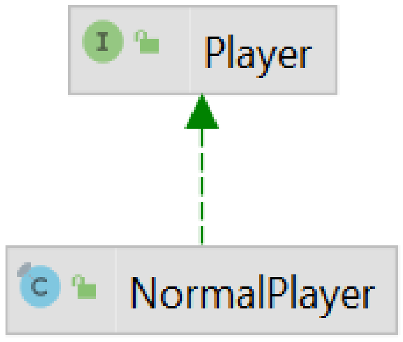
For generalization it can be shown as follow:



Where Piece is an abstract class.



Where Game Rules is an interface



Where Player is an interface

Design Patterns

A factory design pattern which used to create the pieces on the cells, the board class uses this factory to create chess pieces when the game start. Also, the chess game class uses this factory when to promote a pawn to a queen.

The second design pattern is the chess game class, the chess game class has an instance of each object (player, game rules, board). The job which the chess game class do is almost the same job the mediator design pattern does. As a result, the chess game class, and the way of interacting with other objects can be considered a mediator design pattern.

The strategy design pattern was an option to implement move behavior but after more understanding of the chess game, it appears the strategy design pattern will add more complexity to the design when the job can be done simply. What is more, the pieces’ move behavior will never change in the future.

Clean Code

The naming

All the names are descriptive and unambiguous. It has a meaningful distinction, searchable. And there are no magic numbers. Maybe some functions names in the code will disagree here because they are too long like “is valid piece to move” and “is rook and king at start place”. Finally, there is no abbreviation and typos. In the end, the name suits the purpose of the variable, function, and class.

Functions

Most functions do one thing, also this thing is related to a person’s view, like function inside “is valid castling” will check if related pieces are at the correct place to validate the castling. This function can be looked like a function do one or two things from my point of view it considers one thing to do which is checking if the pieces are there or not. And not checking each piece with a standalone function for it.

For function arguments, no function needs 4 arguments or more. The code has some functions with three arguments that need to avoid. Also, most of the functions are simple and not complex. Some of the complex functions that exist in the code “is valid castling” and “do Castling”.

Comments

From the book view, most of the comments are bad and there is a little useful comment. Because of this, there wasn’t any comment left in the final code. Even though comments were useful while implementation like to do comments. These comments are deleted after completing the implementation because they become useless.

Return null

From the book view, the code should not return null. the code avoids returning null by throwing an exception as an example of this in the code, the factory class would return null in case the wrong piece name. This problem was solved by throwing an exception.

Pass null

There is a case where the null is passed in the function, and this happens when the user tries to move an empty cell. There is no piece passed here since the place is empty. To avoid passing this null value, the cell, which was picked by the user will be passed, then check if there is a piece on the cell.

Classes

The classes are mostly small except for two classes which chess game class, and the normal rules class. For game class one of the reasons why it is long is the get and set methods for the private fields. There are many private functions for normal rules class because the validation takes a long process like validating the castling.

Formatting the class

Uncle Bob’s format is used as much as possible in the classes. As the book mention, the private function should be after the first use, the local variable before the first use. So, to find a private function in the code just downward from the first usage.

Inherit constant

There is no constant inherited in the code.

Dead code

The dead code is avoided as much as possible but since some functions will be not implemented a dead code will be there for sure. Some functions will return true or false only.

Duplication

The duplication in the code is very rare, there is only duplication when throwing an exception. There was a duplication where the same function was reused in different classes, this duplication was solved by creating a class that has a public static function.

Clutter

The code is almost free of the clutter, there is a function or two that isn’t removed in case it will be used in the future since the functions responsible for validation have not been implemented in this version of the code.

Encapsulation Conditionals

Most conditions are encapsulated, this can be noticed in board class and chess game class. For conditions which not been encapsulated, encapsulation may make it more complex from my point of view or it is readable.

Avoid negative conditions

The code agrees and disagrees with this since the negative conditions can be found in some parts of the code.

Avoid long import list

The code agrees with this and can be found in some classes like normal rules class. As an example

```
import java.util.*;
import java.util.regex.*;
```

Effective Java

Static factory

Using static factories over constructors is widely used in the design but some classes do not have them because the constructor is empty. Also, in order not to change the interface, the static factory wasn't used for the chess game class.

Avoid creating unnecessary objects

Reusing expensive objects for improved performance, and that's appeared when checking the pattern for the input string from the user. Also, prefer primitives to boxed primitives which occur in many parts of the code.

But sometimes the code disagrees when creating unnecessary local objects to make the code more readable. Like instead of

```
board.getCell(moveFromCell).getPiece();// this can be used

Piece playerPiece = board.getCell(moveFromCell).getPiece();//more readable
```

Override (to string, equal, hash code)

These functions were overridden in all classes. For how the hash is coded in the classes, it is done using the One-line hash code method - mediocre performance.

Final classes

The book mention, that it is good to make classes final so they can't have subclasses. Like, override a method which responsible for validating the piece move and should not override it to change game rules to cheat.

Minimize the accessibility of classes and members

This is done by making all fields private or protected.

In public classes, use accessor methods, not public fields

Each class in the code has accessor methods to its private fields and there are no public fields. Also, the book mentions that to reach the lowest access level, this is done by deleting not important accessors.
(Getters and setters)

Minimize mutability

This is done in the code by making the most of the fields private, making some of the classes final so they can't be extended. Making some fields final.

Favor composition over inheritance

The code disagrees since it is using inheritance. And from my point of view, inheritance should be used since the inheritance exist.

Prefer interface to abstract class

The design has two interfaces and one abstract. Even though the abstract class can be an interface but from my point of view why should the programmer implement the same function with the same code in all sub-classes when the programmer can write it one time in the abstract class?

Prefer list to array

The code sometimes uses arrays and sometimes lists, so the code agrees and disagrees with this.

Design method signatures carefully

The code agrees with using the naming convention, there is no long parameters list, in the code, the maximum number of arguments found is three.

Using String builder instead of concatenation

String build is used in the function where to build the board as a string in the command line.

Prefer foreach on traditional for

The code disagrees with this since there is no foreach in the code. From my point of view the traditional for gives more control to the code.

Refer to objects by their interfaces

The code agrees with this, like with player interface and game rule interface in chess game class. Also, this appears in objects with their abstract class.

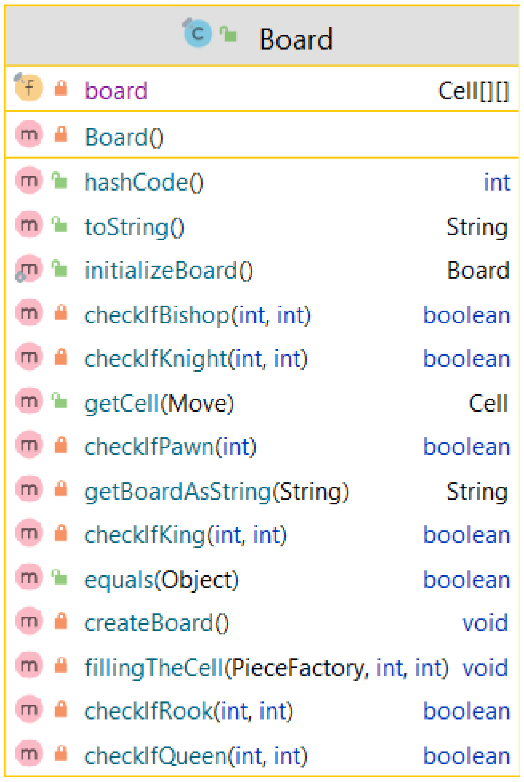
Consistently use the Override annotation

The code agrees with this, and it is obvious in all classes. Almost all the classes have override annotation

SOLID

single-responsibility principle

almost all the classes in the design follow this principle but having big functions inside the class will make the need to make private functions. And that will result in miss leading the class not following this principle. An example where this appears:



Board	
board	Cell[][]
Board()	
hashCode()	int
toString()	String
initializeBoard()	Board
checkIfBishop(int, int)	boolean
checkIfKnight(int, int)	boolean
getCell(Move)	Cell
checkIfPawn(int)	boolean
getBoardAsString(String)	String
checkIfKing(int, int)	boolean
equals(Object)	boolean
createBoard()	void
fillingTheCell(PieceFactory, int, int)	void
checkIfRook(int, int)	boolean
checkIfQueen(int, int)	boolean

This class is responsible for the chessboard, there are many private functions due to some complex functions like create board function, every check function in the class is created to make create board function readable for the user. Also, the get board as string function is related to the tostring function.

The open-closed principle

The design follows this principle in many places like piece class, player class, and game rule class. But this principle may not be applied to in-game chess class because there is no abstract or interface for it. This decision was taken because the client interface is not allowed to be changed.

The Liskov substitution principle

The design agrees with this principle since there is no change in the behavior in subclasses when inheriting the superclass. Like all the subclasses of the piece class does not change the behavior of the superclass. The behavior of the piece should be able to be killed and have white or black color. What is more, the piece should be able to accept or reject the move.

The interface segregation principle

The design agrees and disagrees with this principle, it agrees with the player interface since it is small, but it disagrees with the game rules interface because it is too long. Why not break the interface? Because of the need of using the game rules interface to reference normal rules class without breaking the dependency inversion principle.

Dependency Inversion Principle

The design agrees with this principle since all the classes are not dependent on low-level concrete classes, they depend on abstractions or interfaces. And that is obvious in chess game class and cell class.