# CS 246 Fall 2022 A5 Chess Design Document

Yejoon Jung, Taegwon Kim

## I. Introduction

In the CS 246 Final Assignment, we are given choices of three games to plan and implement using c++ and object-oriented programming paradigm covered in the course, where one of which is the game of chess. During the last two weeks, we planned the design of our chess program, constructed its uml, and built a concrete implementation of the program.

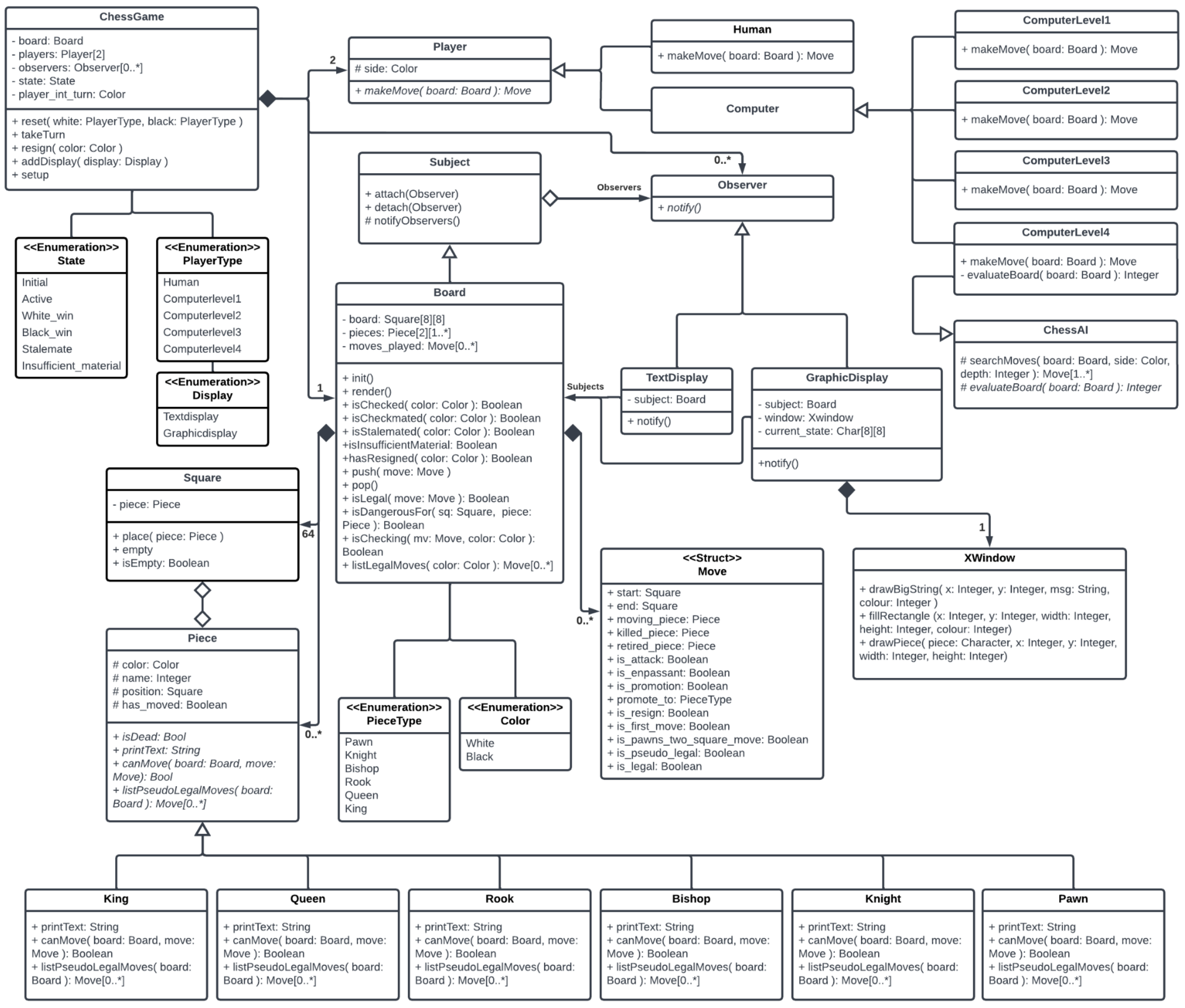
When designing the structure of the program, we made our best attempt to make use of the concepts in the object-oriented programming we newly learned in this course, such as encapsulation, data abstraction, polymorphism, and inheritance. Moreover, when applying those concepts, we focused on achieving the principle of high cohesion and low coupling. Considering those aspects in our program significantly improved the readability of codes and our work experience.

## II. Overview

Our program follows the MVC (Model-view-controller) pattern. The class ChessGame works as the Controller to interact with clients, and the class Board works as the Model to manage the data for representing the chess board. Finally, the class TextDisplay and GraphicDisplay play the role of View to present the chess board to users. These classes interact with one another to perform as a one chess program. ChessGame accepts user requests and modifies Board, and Board notifies TextDisplay and GraphicDisplay to update the user interface and show users the current state of the chess board.

For implementing the Model and View, the observer pattern is used. The class Board inherits the class Subject, and the class TextDisplay and GraphicDisplay inherit the class Observer, whereby the subclasses acquire all attributes and methods required for the observer pattern. In the observer pattern, observers are attached to the subject when constructed, and the subject keeps observers notified whenever needed. In this way, the Board can be observed and presented to users by whichever numbers of displays of different polymorphic types including TextDisplay and GraphicDisplay.

## III. Updated UML


ChessGame
- board: Board
- players: Player[2]
- observers: Observer[0..*]
- state: State
- player_int_turn: Color
+ reset( white: PlayerType, black: PlayerType )
+ takeTurn
+ resign( color: Color )
+ addDisplay( display: Display )
+ setup
2
Player
# side: Color
+ makeMove( board: Board ): Move
Human
+ makeMove( board: Board ): Move
Computer
ComputerLevel1
+ makeMove( board: Board ): Move
ComputerLevel2
+ makeMove( board: Board ): Move
ComputerLevel3
+ makeMove( board: Board ): Move
ComputerLevel4
+ makeMove( board: Board ): Move
- evaluateBoard( board: Board ): Integer
ChessAI
# searchMoves( board: Board, side: Color, depth: Integer ): Move[1..*]
# evaluateBoard( board: Board ): Integer
0..*
Subject
+ attach(Observer)
+ detach(Observer)
# notifyObservers()
Observers
Observer
+ notify()
<<Enumeration>> State
Initial
Active
White_win
Black_win
Stalemate
Insufficient_material
<<Enumeration>> PlayerType
Human
Computerlevel1
Computerlevel2
Computerlevel3
Computerlevel4
<<Enumeration>> Display
Textdisplay
Graphicdisplay
1
Board
- board: Square[8][8]
- pieces: Piece[2][1..*]
- moves_played: Move[0..*]
+ init()
+ render()
+ isChecked( color: Color ): Boolean
+ isCheckmated( color: Color ): Boolean
+ isStalemated( color: Color ): Boolean
+isInsufficientMaterial: Boolean
+hasResigned( color: Color ): Boolean
+ push( move: Move )
+ pop()
+ isLegal( move: Move ): Boolean
+ isDangerousFor( sq: Square, piece: Piece ): Boolean
+ isChecking( mv: Move, color: Color ): Boolean
+ listLegalMoves( color: Color ): Move[0..*]
Subjects
TextDisplay
- subject: Board
+ notify()
GraphicDisplay
- subject: Board
- window: Xwindow
- current_state: Char[8][8]
+notify()
1
XWindow
+ drawBigString( x: Integer, y: Integer, msg: String, colour: Integer )
+ fillRectangle (x: Integer, y: Integer, width: Integer, height: Integer, colour: Integer)
+ drawPiece( piece: Character, x: Integer, y: Integer, width: Integer, height: Integer)
Square
- piece: Piece
+ place( piece: Piece )
+ empty
+ isEmpty: Boolean
64
Piece
# color: Color
# name: Integer
# position: Square
# has_moved: Boolean
+ isDead: Bool
+ printText: String
+ canMove( board: Board, move: Move): Bool
+ listPseudoLegalMoves( board: Board ): Move[0..*]
0..*
<<Enumeration>> PieceType
Pawn
Knight
Bishop
Rook
Queen
King
<<Enumeration>> Color
White
Black
0..*
<<Struct>> Move
+ start: Square
+ end: Square
+ moving_piece: Piece
+ killed_piece: Piece
+ retired_piece: Piece
+ is_attack: Boolean
+ is_enpassant: Boolean
+ is_promotion: Boolean
+ promote_to: PieceType
+ is_resign: Boolean
+ is_first_move: Boolean
+ is_pawns_two_square_move: Boolean
+ is_pseudo_legal: Boolean
+ is_legal: Boolean
King
Queen
Rook
Bishop
Knight
Pawn
+ printText: String
+ canMove( board: Board, move: Move ): Boolean
+ listPseudoLegalMoves( board: Board ): Move[0..*]

## IV. Design

### 1. Design Ideas

There are two important classes consisting of the ChessGame: Player and Board. In real life, as a 2-player board game, the game of chess consists of two players and one chess board. Similarly, we designed our program so that the ChessGame owns two objects of class type Player and one object of class type Board. To support various types of players, we defined the Player as an abstract class, which can be inherited by concrete subclasses such as Human, and ComputerLevel1 to ComputerLevel4. Given the Board, those players select a move from a list of legal moves according to their own algorithms. In chess, the board is comprised of 64 squares and 6 different types of pieces with colour either black or white. Each square is either vacant or occupied by a single piece. We designed our program to reflect those properties. The Board owns 64 objects of class type Square and an unknown number of objects of class type Piece. Each object of Square can hold maximum one of the objects Piece, and also Each object of Piece can hold maximum one Square as its position. The Piece is an abstract class inherited by its subclasses representing and implementing specific behaviours of six different types of chess pieces.

Move in chess is represented by short letters containing its moving piece, start and end position, and other necessary information, which is called the algebraic notation. In our program, we defined a struct Move which contains all the necessary information to do and undo its corresponding moving action on the Board. The method push() in Board consumes an object of Move and makes that move in effect if the move is legal, while the method pop() undoes the last move made on board.

As such, the design of our program resembles the game of chess in real life, and this significantly increases the cohesion of the program. Also, the dependencies between the classes in our program are predictable and minimised, which lowers the coupling of the program.

## 2. Chess Rule Implemented

The Board is the core part for implementing chess since it contains many important methods defining the fundamental rules of chess, including detecting the check, checkmate, stalemate, legal moves, and so on. The fundamental rules and terms of chess we defined for our program are as following:

- **Checked:** when at least one of the **opponent's pieces can move** to the square where the **player's king** is located.
- **Pseudo-Legal Move:** move that its **moving piece can move** from its start square to end square according to its own movement rule
- **Legal Move:** move that is **pseudo-legal** and does **not** leave the **player's king in check**
- **Checkmated:** when player is currently **checked** and have **no legal move**
- **Stalemated:** when player is currently **not checked** and have **no legal move**
- **Insufficient Materials:** when both players are left with either a sole king or a king with a minor piece (bishop or knight).

The Board is equipped with methods to detect and react to all the rules and concepts in chess defined above. For example, methods isChecked(), isCheckmated(), isStalemated() return a boolean value given a colour of player, depending on current board state. Also, given a Move, isLegal() detects whether it is legal or illegal. The methods isDangerousFor() and isChecking() are utility functions used by computers to select moves (and also by hidden implementation of other methods in Board).

When checking legality of moves, we first have to look at its pseudo-legality. However, the movement rules are different by the types of chess pieces, and it is against the object-objected programming principles to cover every case in a single function via if statement. To resolve this problem, in our program, the pseudo-legality of moves are checked by the method canMove() at the corresponding subclass of Piece objects. In this way, methods isLegal() can call the canMove() function at subclasses of Piece representing the type of chess piece corresponding to its moving piece to check the pseudo-legality, and then check whether the move puts the player's king in check.

Finally, the method listLegalMoves() returns a vector list of all legal moves available to the player, given the colour of the player. This is used for giving possible choices of moves to the computer players. It works by generating all the pseudo-legal moves by calling listPseudoLegalMoves() from each of alive pieces with the player's colour, looping over the generated moves, and pushing each move into the resulting vector list if it passes the legality test.

### 3. Changes from the Initial Plan

Although there are not much difference between the initial and final uml of our program, you might notice a few changes. Firstly, the struct Config has been deleted on the final uml. Initially we thought it is required for implementing the setup feature, but as we progressed we found that it is not necessary. Secondly, Move has been changed from class to struct. This is because as we write the codes, we found that Move does not necessarily need to be encapsulated since there are no particular class invariants for Move, and defining all the getters and setters for Move made the code less readable (this will be covered again in the final question). Thirdly, a new class ChessAI was created as a parent class of ComputerLevel4. This class is for the features for an extra credit and will be covered again later. Lastly, a few more enumerations were defined to represent types of player, display, and game state.

## V. Resilience to Change

Utilising data abstraction and polymorphism, our program was designed to be very resilient to later changes. Let’s consider different types of changes that might be required in the future.

### 1. More Types of Player

In order to add more types of player in the game, we can simply build a new class inheriting the class Player (if it is computer then inherit the class Computer) and override the virtual method makeMove() according to its own algorithm of selecting a move. After that, we can register the player to the controller ChessGame by adding a new value in the enumeration PlayerType and a few lines of necessary codes.

### 2. More Types of Piece

If we want to add more types of piece that move differently from the regular standard chess pieces, we can build a new class inheriting the class Piece and override the virtual functions canMove() and listPseudoLegalMoves(). After that, we can register the piece to the Board by adding a new value in the enumeration PieceType and a few lines of necessary codes.

### 3. Larger Board

Numbers related to the dimensions of the board are declared as a constant in the class Board and Displays. Thus, In order to enlarge the board, we can change the numbers in the constants and make other minor required changes.

### 4. Change in Rules

The rules of chess in our program are subdivided and implemented as methods of the class Boards. In order to change a specific part of rules, we can make changes accordingly only to the methods where that specific part is covered.

## VI. Answers to Questions

**Question 1:**
***Chess programs usually come with a book of standard opening move sequences, which list accepted opening moves and responses to opponents' moves, for the first dozen or so moves of the game. Although you are not required to support this, discuss how you would implement a book of standard openings if required.***

Given a set of standard opening move sequences, we can implement a class representing a book of standard opening, which reads and translates the sequences into a single tree structure with each node containing the struct Move, where all the opening moves are arranged in a way that a player can navigate through in a constant time. We can use this class to build new computer players that prioritise the opening moves whenever possible or to help human players to find better moves.

**Question 2:**
***How would you implement a feature that would allow a player to undo their last move? What about an unlimited number of undos?***

The struct Move, representing each move played in the game of chess, contains the information of its corresponding move including the start position, end position, piece moved, piece killed, and whether the move is any of castling, promotion, or en passant. Whenever a player makes a Move, Board records that Move by pushing it into a vector moves_played. This allows players to undo their moves unlimited times (until there are no more moves to undo) by reverse operating each move and popping it out of the vector moves_played one by one from the back. Specifically, when undoing the last move, the Board goes through the following steps (with special moves such as castling must be handled appropriately with additional steps): get the last element of moves_played, retreat the piece moved by the last Move backward to its start position, if there is a piece killed by the Move then revoke the piece to the end position, and finally pops the last move out of the vector moves_played. In fact, we have already implemented this feature in our program for an extra credit.

**Question 3:**
***Variations on chess abound. For example, four-handed chess is a variant that is played by four players (search for it!). Outline the changes that would be necessary to make your program into a four-handed chess game***

In order to accommodate the four-handed chess game, we first need to increase the number of players of different colours from 2 to 4. It can be done by editing constants and enumerations defined in ChessGame and Board, which will automatically update the array of Players to length 4. Next, we need to make changes on the dimension, initial configuration, and certain methods of the board so that it matches the rule of four-handed chess. Also, we need to make changes to the ChessGame and command interpreter in the client main.cc, so that it can support the commands for four-handed chess games. Finally, we need to make changes on the TextDisplay and GraphicDisplay classes so that it can correctly display the newly configured board with expanded board size and pieces of two newly added colours.

## VII. Extra Credit Features

### 1. Takeback

We have implemented the takeback feature for the human player. It can be used during the game at the human player's turn by entering the command "takeback" into the prompt. When used, the last two moves are revoked so that the human player can have a chance to change his last move. Implementing this feature was challenging because it was tricky to find how  to reverse operate a given move.

### 2. ChessAI

In this program, we have implemented a chess AI using MiniMax algorithm with alpha beta pruning to support the custom level computer player ComputerLevel4. In the class ChessAI, white player is defined as a maximising player, and the black player is defined as a minimising player. Given a current state of board, its function evaluateBoard() returns an integer value determining which player is more likely to win the game in how much degree. A positive value means white player is winning, and a negative means the opposite. Also, the higher the value is, the more the white player is likely to win, and vice versa. Given Board, colour of player, and depth for searching, The function searchMove() in the class ChessAI returns a vector list of moves that result in a board state that produces the most favourable value from evaluateBoard() for the player of the given colour after the given depth of possible following moves played according to the Minimax algorithm. The function evaluateBoard() was declared as virtual in the class ChessAI since the difficulty of AI can be adjusted by the quality of the evaluateBoard() function. The class ComputerLevel4 inherits both (multiple inheritance used) Computer and ChessAI, and overrides the virtual methods makeMove() and evaluateBoard() from both parent classes. Then it uses the method searchMove() from ChessAI with its own overridden evaluateBoard() method to generate the candidates of moves to play.

Implementing and debugging the Minimax and alpha beta pruning was challenging since their concepts are quite confusing and algorithms are hard to track with a human's brain. So when we were implementing those, we tried to understand and stick to the logic of algorithms rather than the actual output of the function.

## VIII. Final Questions

**1. What lessons did this project teach you about developing software in teams? If you worked alone, what lessons did you learn about writing large programs?**

Working as a group, we realised the importance of initial planning. When working individually, planning is less important because one can be flexible as much as he wants and constantly change the design of the program while implementing the program. However, when working as a group, being flexible on the general design of the program can cause troubles in many situations especially for large programs since you have to notify and explain the changes every time to all group members so that they can also make changes accordingly on their parts, which is very confusing and inefficient. On the other hand, if we have a well designed plan of a program clearly specifying its structure and class interfaces and always stick to it, we can prevent all the possible confusions and hugely improve our efficiency and work experience.

**2. What would you have done differently if you had the chance to start over?**

Although we would like to keep the original design in most parts of the program, we want to improve it in two ways if we were given a chance. First, we would like to further optimise the performance of Board and ChessAI by reconstructing the control flow and logics of chess rules implemented in the program. Compared to other chess engines found online, our Chess AI is a bit slower. We want to identify the cause of this difference and improve its performance if we were given more time. Second, we would like to change the struct Move back to class. We mentioned earlier above that Move has no particular class invariants. However, while debugging the program, we realised that there are  actually a few invariants that would make the program less error-prone by making its possible bugs easily detectable. For example, one invariants would be that the killed_piece must be setted only if is_attack is true.