Chess Final Design

1.Overview

In our final design, there are three main components: Board, Game and Piece. Board is the part which provides visualization including both text-display and graphical display. Game is the controller of the entire program. It not only controls how to move the pieces but also when the Board needs to visualize it. Piece is a class where it has all information and the methods about the given piece. Furthermore, the whole program is based on the interaction between Game and Piece and their behaviour. As for players of the game, there are 5 types of players: Human, Computer I, Computer II, Computer III and Computer IV. Human players play the game via standard input while Computer players play the game via their reactions to a given situation based on its different level of logic implementation.

2.Design Details

I Explanation of design pattern:

The main design pattern we used is MVC(**Model-View-Controller design pattern**). More specifically,
Model: data, which is the state of the pieces and the state of Game. Every Piece has some information, e.g. its colour and type.
View: Board, which controls two types of display: (1)text (2)graphical
Controller: Game is the controller which receives the input from the user and can work with Board and Piece. Game implements users' commands if feasible and decides whether or not to call Board to display it. Moreover, Game interacts with every Pieces and changes their states.

II Changes from the original Design

First, we noticed that there were many similarities between Chess and Reversi so that we wanted to imitate the observer pattern used in Reversi. However, the situation is much more complicated in chess since we have more types of pieces and we feel that using the observer pattern may actually complicate the implementation of our program. Therefore, we decided to construct a structure called Pieceset that includes all the information about the pieces. Every piece is given the address of Pieceset; hence, each piece can access all information and update of other pieces.

Second, we delete the class called Player and all subclasses in the previous UML. At first, we want to separate human and computer players. But later we noticed that they had many similarities and it is easier to implement under the class Game. Therefore, we decide to put them in the Game class.

III Cohesion and coupling

(1) Our program is of low coupling

As shown in the new UML, we separate the three aspects of the program into three parts. View, controller and model. View includes two types of display methods(text-display/graphic display). A controller is implemented in the game class. Model is the logic and rules of chess pieces which are implemented in the Piece and Pieceset. Hence, our program depends on three different parts which work relatively independently. For example, in our Game class, we do not need to decide which positions are valid for a given chess piece; instead, they are calculated using the method validpos() in each chess so that Game can take these valid positions as correct and granted. Furthermore, there is no dependency between the chess piece and the View. Game calls the View only when Board states is changed, and they work independently. In conclusion, our program is of low coupling.

(2) Our program is of high cohesion

In view, we have three different states: create, tDisplay and gDisplay with given Pieceset. They are all necessary to achieve the functionality of View. Hence, they are all important and there are no redundant methods or fields. In terms of the controller, Game is in charge of setting up a new game which could be a standard new game consisting of 32 pieces or a user-defined setting up which could contain any number of any type pieces. In addition, Game is responsible for determining legal moves and updating piece and Board states, which requires full knowledge of information of all pieces. Therefore, it is necessary for the Game to carry all information of pieces so that it can decide whether a move a legal. Piece implements all the necessary methods that can obtain and mutate its own fields and also calculate its valid positions and capture positions (valid positions where an enemy piece locates and can be captured) for Game to implement its own methods. Although Game and Piece work independently, they have a lot of relationships and share a lot of information.

3. Resilience to Change

Our design supports various changes to the program specification. Since we use the MVC pattern and it is easy to locate where we need to change in our code. For example, if we want to change the rules that control the moves of chess pieces, we only need to change the code in Pieces and change their behaviour accordingly. If we want to change the way to display the game, we need to change the code in Board which provides both text-display and graphical display and replace them with new methods to display. Moreover, if we want to change the way how computer players behave, we only need to modify the code in Game and add our new implementation. Therefore, our program has some resilience to change if the change is not too significant.

4.Answers to Questions (the ones in your project specification)

Q1: 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.

Previous answer: Construct another module named "Standard Opening." Which stores some general opening patterns and their counter plays. After the opponent played, if the opponent's play matches the pattern, the counter play will be done using a specific pattern "Standard Opening."

New answer: The new answer is quite similar to the previous one with a few differences. We will construct another module named "Standard Opening", which stores a few vectors of Move structures. Each vector corresponds to a particular standard opening. When two human players are playing a match, the users have an option to invoke a particular standard opening. After that, the program will apply the chosen standard opening and the board state will adjust automatically as if the Moves in the chosen standard opening were made accordingly. Instead of invoking the standard opening directly, the user could input the keyword "hint", then the program will search all the standard openings and suggest the next move to the user.

Q2: How would you implement a feature that would allow a player to undo his/her last move? What about an unlimited number of undo?

Previous answer: As illustrated in the UML, the class "Game" has a field called "past

Moves," which stores all historical moves and a method "addMove()," which adds new moves to the historic moves to "pastMoves." Every time we need to undo one time, we access the last element of "pastMoves," restore to the previous state and pop out the last Move.

New Answer: the same as the previous answer, as implemented in our program.

Q3: 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.

Previous answer: First, we need to modify the board to fit the four-player mode, and increase the colour types from black and white to four colours. Also, the "isWhite" Boolean in class "Game" need to change to "currentColor," indicating the current player. Also, now each chess piece needs to observe all three player's chess pieces, increasing the size of its field "observers."

New answer: the same as the previous answer.

5. Extra Credit Features

Feature1: Unlimited Undo

Description: this feature is challenging because when we undo a step, we need to consider many things: (1)how to record the move (2)how to recover the chess pieces that have been captured and the chess pieces that capture other chess pieces (3)what if we have already promoted a pawn (4)what if we have used the castling.

Solution: We use a particular field called pastMoves which is a vector of Move(also a structure). It records all the valid moves that we have made, and in each move, we know which chess piece(s) is/are changed; hence, each time we want to undo, we pop the last Move in pastMoves and locate the chess pieces which has been moved. And we can decide whether there are chess pieces that have been captured. If the chess piece is King or Pawn, we have special fields to record whether castling or promotion has been implemented. Therefore, we can undo successfully for unlimited times.

Feature2: Computer Level 4

Description: we need to write a strategy which is more sophisticated than computer level 3(prefers avoiding capture, capturing moves, and checks)

Solution: Based on computer level 3, we add different weights to each chess pieces such that lager weights indicate more importance while small weights indicate less importance. The weights are given below:

King: 8848

Queen:9

Rook:5

Bishop:3

Knight:3

Pawn:1(after it is promoted, its weight is changed to be the weight of whichever type it is promoted to)

Therefore, we want to capture chess pieces with large weights, avoid large weights of chess pieces been captured and checks. Hence, it is smarter than computer level 3.

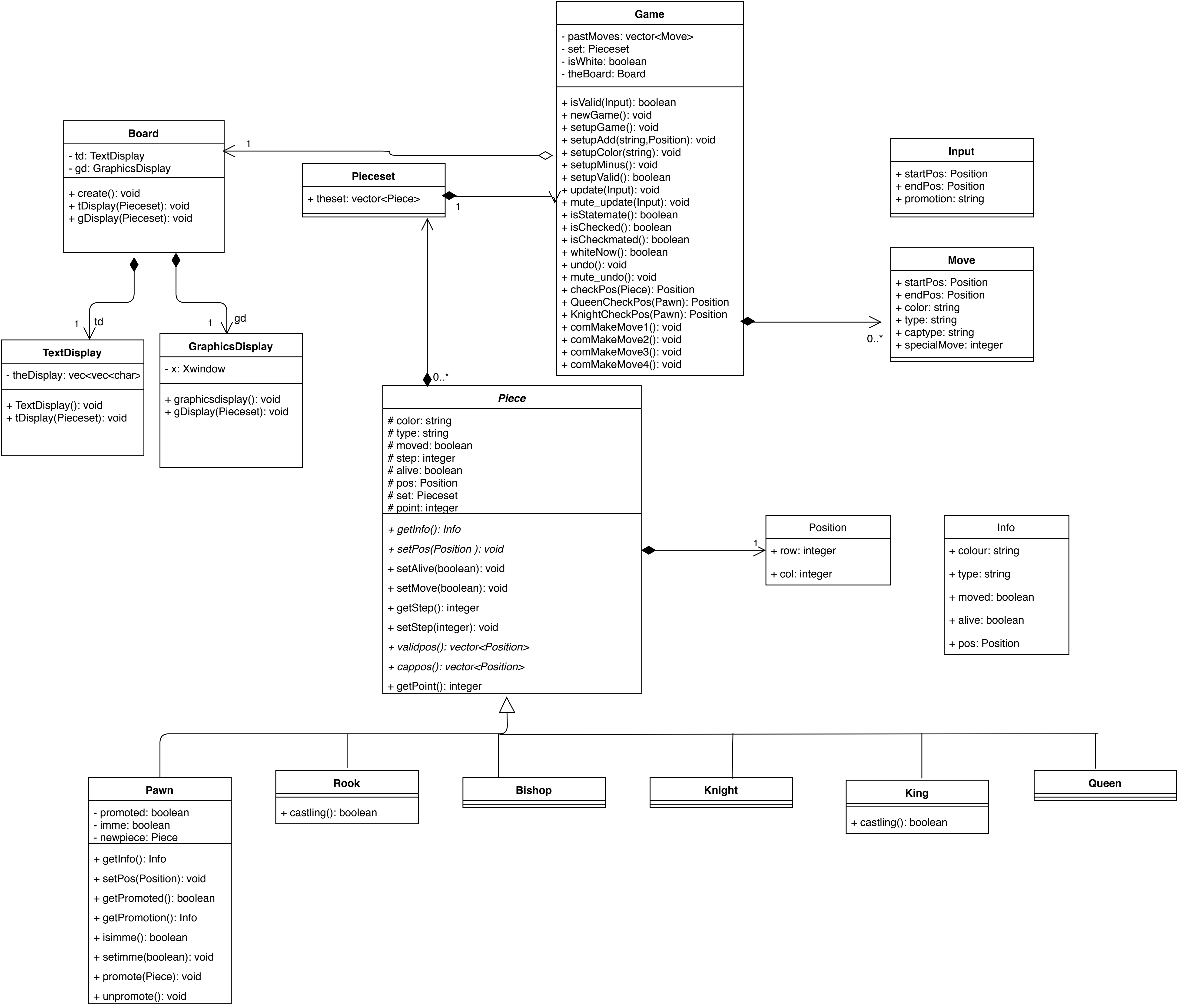
6. Final Questions (the last two questions in this document).

Q1: 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?

Answer: Working in a group is quite different from working individually. Communication and leadership play a key role during the development. Every teammate not only needs to write his own code but also understand others' code. Therefore, each member must be willing to explain his code to other teammates patiently and clarify any misunderstanding. In order to achieve this goal, everyone is supposed to write neat code and add necessary comments to explain the functionality of certain blocks of code. Essentially, everyone needs to collaborate with each other to make the code actually cohesive and executable.

Q2: What would you have done differently if you had the chance to start over?

Answer: We will definitely spend more time on designing our program and coming up with a better and more thoughtful UML. Initially, we wanted to write the UML as soon as possible so that we could start writing code immediately. However, as we started building our program based on our initial UML, we noticed that our original plan did not work as it intended. So it actually took us much more time to redesign our UML and rewrite a lot of code which makes our first 2 days' coding meaningless.


Game
- pastMoves: vector<Move>
- set: Pieceset
- isWhite: boolean
- theBoard: Board
+ isValid(Input): boolean
+ newGame(): void
+ setupGame(): void
+ setupAdd(string,Position): void
+ setupColor(string): void
+ setupMinus(): void
+ setupValid(): boolean
+ update(Input): void
+ mute_update(Input): void
+ isStatemate(): boolean
+ isChecked(): boolean
+ isCheckmated(): boolean
+ whiteNow(): boolean
+ undo(): void
+ mute_undo(): void
+ checkPos(Piece): Position
+ QueenCheckPos(Pawn): Position
+ KnightCheckPos(Pawn): Position
+ comMakeMove1(): void
+ comMakeMove2(): void
+ comMakeMove3(): void
+ comMakeMove4(): void
Board
- td: TextDisplay
- gd: GraphicsDisplay
+ create(): void
+ tDisplay(Pieceset): void
+ gDisplay(Pieceset): void
1
Pieceset
+ theset: vector<Piece>
1
Input
+ startPos: Position
+ endPos: Position
+ promotion: string
Move
+ startPos: Position
+ endPos: Position
+ color: string
+ type: string
+ captype: string
+ specialMove: integer
0..*
1
td
1
gd
TextDisplay
- theDisplay: vec<vec<char>
+ TextDisplay(): void
+ tDisplay(Pieceset): void
GraphicsDisplay
- x: Xwindow
+ graphicsdisplay(): void
+ gDisplay(Pieceset): void
0..*
Piece
# color: string
# type: string
# moved: boolean
# step: integer
# alive: boolean
# pos: Position
# set: Pieceset
# point: integer
+ getInfo(): Info
+ setPos(Position): void
+ setAlive(boolean): void
+ setMove(boolean): void
+ getStep(): integer
+ setStep(integer): void
+ validpos(): vector<Position>
+ cappos(): vector<Position>
+ getPoint(): integer
1
Position
+ row: integer
+ col: integer
Info
+ colour: string
+ type: string
+ moved: boolean
+ alive: boolean
+ pos: Position
Pawn
- promoted: boolean
- imme: boolean
- newpiece: Piece
+ getInfo(): Info
+ setPos(Position): void
+ getPromoted(): boolean
+ getPromotion(): Info
+ isimme(): boolean
+ setimme(boolean): void
+ promote(Piece): void
+ unpromote(): void
Rook
+ castling(): boolean
Bishop
Knight
King
+ castling(): boolean
Queen