CS246 （Winter 2019）： The Game of Chess

Design Document
Qiushi Du(q6du)
Yuqiao Sun(y427sun)

Introduction:

We developed this game of chess in CS246 of University of Waterloo as our final project. This game consists two players and six different types of pieces on an 8 x 8 checkboard, each player is represented by a color: Black or White. A piece captures another piece by moving into the square and occupied by that piece. The piece then is permanently removed from the board. The object of this game is to place your opponent's king under attack, if it has no where to escape, you win the game. Details of the game are outlined in the assignment guideline.

Overview:

Our gaming implementation consists two level of difficulties for the graphical interface and computer AI using X11. It can implement all three of Human vs Human, Human vs Computer and Computer vs Computer. All functionality from assignment guideline can be implemented except castling, promotion and level 4 computer implementation. Below, we will talk about our design thoughts and attach some documentations as instruction for playing our games.

Design:

we want to implement a module that cost less, which means we want to avoid too much of copy of parameters to be made. Therefore, we construct some public methods using pointers, passing pointer instead of value would make our program less expensive. (This is implemented in Classes Coordinate, Piece and Checkboard).

Class: Piece

Piece is an abstract class. It has six different pieces that used to be reflected on the Checkboard. This class is helpful for us to organize the subclasses using pointers. Whenever the piece is moved or removed from the board, the Piece in this class is modified first, then get updated on the board.

+ canmove(int r, int c): it will determine whether or not a piece have space to move

+ getname(): get value of the field name, used to determine it color

+ getvalue(): get value of the field value, used in graphic display later

+ getcolor(): get color of the piece

+ getcoor(): get the square that the piece is at

+ setcoor(): set the value to the square of the piece

+ hasmoved(): return true if it has been move, vice visa.

+ moveto(Coordinate *move): move the piece to other place

+ undo(): move piece to its last position, also used in canmove() to determine whether a piece can move or not.

Class: King, Rook, Queen, Knight, Bishop, Pawn

King on board can move in any direction in one space, capture it to win and inherits from piece.

Queen on board can move in any direction in as many spaces as it wants and inherits from piece.

Rook on board can move in horizontal and vertical direction in as many spaces as it wants and inherits from piece.

Pawn on board can move in any direction in as many spaces as it wants and inherits from piece.

Bishop on board can move in diagonal direction in as many spaces as it wants and inherits from piece.

Knight on board can displace itself three units in any combo of left, right, up and down, it can jump over pieces. Inherits from piece.

Class: Coordinate

Coordinate is new class that we added after the first version of UML, we decided to split the original class Piece into two: Piece and Coordinate. Piece is in control of all pieces and Coordinate is the class that reflects everything about the field.

Coordinate has a Piece class pointer and some other features, which implement a single coordinate field in the board.

+ getc(): get column number of a field

+ getr(): get row number of a field

+ getx(): get x value of a field

+ gety(): get y value of a field

+ setoccupied(): set the parameter to be true, which means it has been occupied by other piece

+ hasoccupied(): if the field has been occupied, return true, vise visa.

+ getoccupied():if occupied, return the occupied piece

Class: Subject

Subject is an abstract base class for subjects

+ notifyObservers(): notify to update the checkboard once move is completed

Class: Checkboard

Checkboard implements a 2-dimensional field with pieces on it, along with players and some other features needed.

+ init(): set the initial position of both side of pieces, and get them a color.

+ clear(): clears the board, and update a new board

+ setTurn(): determine which player's turn

+ attatchCommand(Command &c): record the command the player entered and make it useful in the function printhistorymove().

+ check(string king): return true if king is in check, vise visa.

+ underattack(Piece *p): determine whether a piece is under attack or not.

+ printmmoves(): print all historical moves

- stalement() and checkmate() will determine the result of the game.
- setup() and unset() will restart the game.
- Clearvertical(), clearhorizontal() and cleardiagonal() checked whether it is clear in the required direction.

Class: Observer

We used observer pattern to display the board in both graph and text, so after user moves, notifyObserver() in Coordinate object will update the checkerboard.

Struct: Command

Command is a structure that provide representation of the move command. It contains four elements: a vector that records the move, a Piece pointer that points to the captured pieces and a Boolean value that shows whether or not pawn is promoted.

Class: Display (TextDisplay/ GraphicDisplay)

+ init(): set the initial textdisplay or graphicdisplay

+ notify(Subject &whoNotified):

Class: Player (Human / Computer)

The abstract class player consists two derived class: human and players, which support both computer and human moves.

+printnumplayer(): print the number of players

+winscore(): make score add up by 1 if anyone won the game

+draw(): each of the player would get 0.5 points

<u>High Level Implementation:</u>

Level 1: In this level, computer doesn't have any preferred strategies on how to attack or defend pieces, the move will just be random legal move. Making use of canmove(), computer need to first detect whether there is a space available for them to move, if doesn't, function undo() is used to undo the movement,

Level 2: Since we have defined every piece a different value, for example, king will have the value of 900, which is the biggest among others, and other pieces have other value. At this level, computer tends to attack pieces that have a larger value if there is space available. If the piece cannot attack the "biggest" valued piece, undo the movement, then try the same thing with the piece that has second "biggest" value.

Level 3: Similar as level 2, computer at this level prefers to defend pieces, so at this level computer tends to avoid capture, capturing and checks. It would detect which piece would be under attack at the next round and move it to a safer place.

Bonus Features:

- ***Undo***: Having player to undo his last move or unlimited undo the move, player can simply call “undo” and board would use function undo() to undo the last move. Undo features is also use in the function canmove(), it detects whether it can move to that square or not, if the move cannot be completed, we will use undo() to decline the movement.
- ***Printing History Moves:*** we have created a public method: printmoves() in the class Checkboard. This function will print all the commands that the two players typed, by accessing the Command structure in the main function.

Updated UML:

Our final design and our original UML differ in many ways. I will specify the changes in each class below. There are three new classes that is added into the implementation: Coordinate, Subject and Observer. Updated UML is attached as an appendix at the back.

In *piece.h*:

Fields are removed:

+ notify(Textdisplay *td): void

+ setcoords(int r, int c): void

+ getrow(): char

+ getcolumn(): char

+ indanger(): int

+ istype(string str): bool

+ setpiecescolors(string white, string black): void

Fields are added:

+ getname(): char

+ getvalue(): int

+ restore(): void

+getcolor(): sting

+getcoor(): Coordinate

+ moveto(move: Coordiante): void

+ undo(): void

+ castling_move(move: Coordinate): void

In all *queen.h, bishop.h and knight.h*:

Fields are added:

+ getname(): char

+ getvalue(): int

In *rook.h*:

Fields are removed:

+ setmoved(): void

Fields are added:

+ getname(): char

+ getvalue(): int

+ undo: void

+ castling_move(move: Coordinate): void

In *pawn.h*:

Fields are removed:

+ incheck(): bool

+ moved(): bool

+ ablepromo(): void

+ isenpassant(): bool

+ setenpassent(): void

+ unsetenpassent (): void

Fields are added:

+ getname(): char

+ getvalue(): int

+ moveto(move: Coordiante): void

+ restore(): void

+ abs(n: int): int

+ undo(): void

+ isenpassent(move: Coordinate): bool

In *king.h*:

Fields are removed:

+ incheck(): bool

+ checklegal(): bool

+ setmoved(): void

+ checklegal(): bool

+ setchecked(): void

+ setunchecked(): void

Fields are added:

+ getname(): char

+ getvalue(): int

+ moveto(move: Coordiante): void

+ restore(): void

+ abs(n: int): int

+ undo(): void

+ iscastling(move: Coordinate): bool

In *checkboard.h*:

Fields are added:

+ attatchCommand(c: Command): void

+ getCommand(): Command

+ clear(): void

+ undo(): void

+ setturn(t: string): void

+ printmoves(): void

+ move1 (color: string) : void

+ verify(): bool

+ underattack(p: Piece): bool

+ stalemate(color: string): int

+ standardopening(color: string): void

+ move2 (color: string) : void

+ move3 (color: string) : void

+ move4 (color: string) : void

In *player.h*:

Fields are added:

+ getIs_com(): bool

+ winscore(): void

In *graphicsdisplay.h*:

Fields are removed:

+ fillBoard(int x, int y, int width, int height, int colour = Black) : void

Fields are added:

+ notify(whoNotified: Subject): void

Resilience to Change:

As we discussed in the design of the game, different classes serve different functionalities. A change to one of the classed will not cause some major changes to the other implementation. For example, public methods moveto() and canmove() are implemented in the class Piece, if the rule of the game changed, or some pieces need to move in a different way, then header files have no need to change and by only making change in .cc file, we are able to implement the requirement, which represent high cohesion of files.

So the design of the game of chess achieved low coupling and high cohesion.

Questions:

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.

First, we will use the function canmove(Coordinate *move) to detect whether there is a space that the piece can possibly move, since this function has been implemented to all pieces, we can detect all possible result from the function. Since this function returns a Boolean, if it returns a true, we will apply apply the idea of computer level 1, 2, and 3 to this question. If the possible move can bring profit (such as putting the opponent's king in check), we may suggest this move to the player.

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

To support undo and unlimited undo feature, we decide to use a vector which records every move of pieces. This avoids solving the problem of recovering a deleted piece because by doing this we are deleting all pieces when the entire game is finished. When a player wants to undo the moves, we use the information in our move record vector and apply it to the move function. Additionally, we will add another field in piece called deleted as a bool to mark the piece as deleted or not. Having player to undo his last move or unlimited undo the move, player can simply call "undo" and board would use function undo() to undo the last move.

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.

As for the chances that the game may be put into a four-handed game, first we will need to create two more player, and will need to have an extra three rows or columns to each side. So the size of the board and the way we initialize the board will be changed. Also, in this case the four players will be independent, which means once a player is removed from the game, the game is still on and no other player needs to be removed.

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?

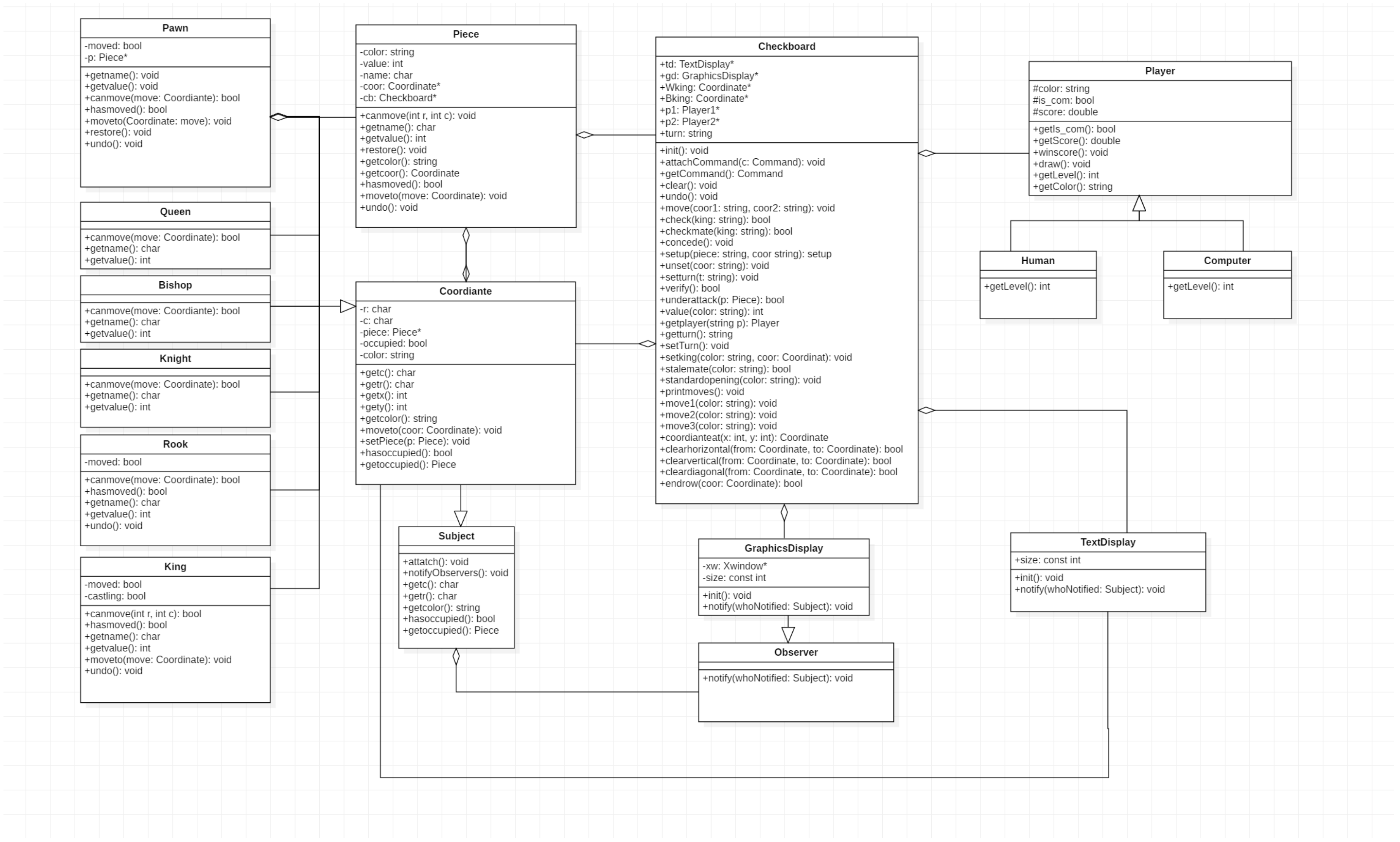
This is the first time that we have the chance to work together as a team to develop a game, I can feel there are some differences comparing with working by myself. There are lots of advantages to develop software in teams. For example, brainstorming as a team is better and quicker than thinking alone. Although at the beginning of developing this game, we had a hard time come up with how the structure of the game should be like, but we finally had an idea before the DD1 with decent quality, so that we can use that structure and idea to implement the game. Also, working together has an excellent efficiency when it comes to debugging. Since this project is a fairly large program, it is hard to debug alone, but are able to discuss the obstacles we faced, and each of us is responsible for a little piece of code, debugging become easier for us.

If I worked alone, I think learn more about how to structure a large program, because if I am writing all by myself, I have no one to discuss, therefore I will gain more knowledge on how to design the structure and connect every individual classes together.

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

Since we have a hard time debugging during the project, so we realized how important test cases are. So if we had the chance to start over, we would build a public structure first, and write some public tests, to make sure our code is partially functional. Also, we would include a more rigorous timeline, we would communicate more and have more group meeting to discuss our ideas towards each part of the code, this would save us a lot of time with debugging. Having a more compact timeline is also benefit for the quality of the game, because we can try to achieve all the features included in the guideline first, including promotion, castling and en passent, and had more time at the end to add some bonus features towards our game.

This is the updated UML:


Pawn
-moved: bool
-p: Piece*
+getname(): void
+getvalue(): void
+canmove(move: Coordiante): bool
+hasmoved(): bool
+moveto(Coordinate: move): void
+restore(): void
+undo(): void
Queen
+canmove(move: Coordinate): bool
+getname(): char
+getvalue(): int
Bishop
+canmove(move: Coordiante): bool
+getname(): char
+getvalue(): int
Knight
+canmove(move: Coordinate): bool
+getname(): char
+getvalue(): int
Rook
-moved: bool
+canmove(move: Coordinate): bool
+hasmoved(): bool
+getname(): char
+getvalue(): int
+undo(): void
King
-moved: bool
-castling: bool
+canmove(int r, int c): bool
+hasmoved(): bool
+getname(): char
+getvalue(): int
+moveto(move: Coordinate): void
+undo(): void
Piece
-color: string
-value: int
-name: char
-coor: Coordinate*
-cb: Checkboard*
+canmove(int r, int c): void
+getname(): char
+getvalue(): int
+restore(): void
+getcolor(): string
+getcoor(): Coordinate
+hasmoved(): bool
+moveto(move: Coordinate): void
+undo(): void
Coordiante
-r: char
-c: char
-piece: Piece*
-occupied: bool
-color: string
+getc(): char
+getr(): char
+getx(): int
+gety(): int
+getcolor(): string
+moveto(coor: Coordinate): void
+setPiece(p: Piece): void
+hasoccupied(): bool
+getoccupied(): Piece
Subject
+attatch(): void
+notifyObservers(): void
+getc(): char
+getr(): char
+getcolor(): string
+hasoccupied(): bool
+getoccupied(): Piece
Checkboard
+td: TextDisplay*
+gd: GraphicsDisplay*
+Wking: Coordinate*
+Bking: Coordinate*
+p1: Player1*
+p2: Player2*
+turn: string
+init(): void
+attachCommand(c: Command): void
+getCommand(): Command
+clear(): void
+undo(): void
+move(coor1: string, coor2: string): void
+check(king: string): bool
+checkmate(king: string): bool
+concede(): void
+setup(piece: string, coor string): setup
+unset(coor: string): void
+setturn(t: string): void
+verify(): bool
+underattack(p: Piece): bool
+value(color: string): int
+getplayer(string p): Player
+getturn(): string
+setTurn(): void
+setking(color: string, coor: Coordinat): void
+stalemate(color: string): bool
+standardopening(color: string): void
+printmoves(): void
+move1(color: string): void
+move2(color: string): void
+move3(color: string): void
+coordianteat(x: int, y: int): Coordinate
+clearhorizontal(from: Coordinate, to: Coordinate): bool
+clearvertical(from: Coordinate, to: Coordinate): bool
+cleardiagonal(from: Coordinate, to: Coordinate): bool
+endrow(coor: Coordinate): bool
GraphicsDisplay
-xw: Xwindow*
-size: const int
+init(): void
+notify(whoNotified: Subject): void
Observer
+notify(whoNotified: Subject): void
Player
#color: string
#is_com: bool
#score: double
+getIs_com(): bool
+getScore(): double
+winscore(): void
+draw(): void
+getLevel(): int
+getColor(): string
Human
+getLevel(): int
Computer
+getLevel(): int
TextDisplay
+size: const int
+init(): void
+notify(whoNotified: Subject): void