CS246 A5 Chess Design Document

By Jiwook Kim, Nipun Lamba, and Prabhapaar Batra

Overview

As our UML suggests, we've got a Board class that pertains to all the functions of the chess board and its interaction with our observers and subjects. They are 64 cells on the board along with the text and the graphics display. The observers for each subject are set using the setObserver(). In terms of board construction, our program uses init() to construct the chess board and place the cells on it in the form of vector of vector of **Cell**. Afterwards, game_default_setting() places the default setting of the chess pieces in their respective positions. The board class has various boolean values to specify the current state of conditions such as check, checkmate, castle, stalemate and en-passant along with their respective setters and getters.

The other important methods defined in the board class are move() and canmove(). First of all, canmove() takes in the name of the piece with the initial and the final coordinates given in the command interpreter and helps the program to decide if that specific piece can carry out the move. We've used canmove() with various other special conditions to write move() that finally makes the move if valid. Furthermore, move() helps us carry out castling, en-passant, check, checkmate and Pawn promotion. For an invalid move, move() throws an Invalid exception and the user gets a message to try again. There are other exceptions that are thrown; for example, in case of an invalid move during check or attempting to move the opponent's piece.

The other classes that are of major significance are **Cell** and **Piece**. **Cell** is discussed further in the design pattern part of the document. The Piece class has seven children, the 6 pieces in chess: King, Queen, Rook, knight, Bishop and Pawn along with a NoPiece class to represent an empty cell when needed. We have given names and specific integer values to the pieces and have used them to prioritise the pieces. The Piece class outlines all the functions carried out by all various pieces on the board. The program possesses getName(), getValue() getters for the pieces' member variables:

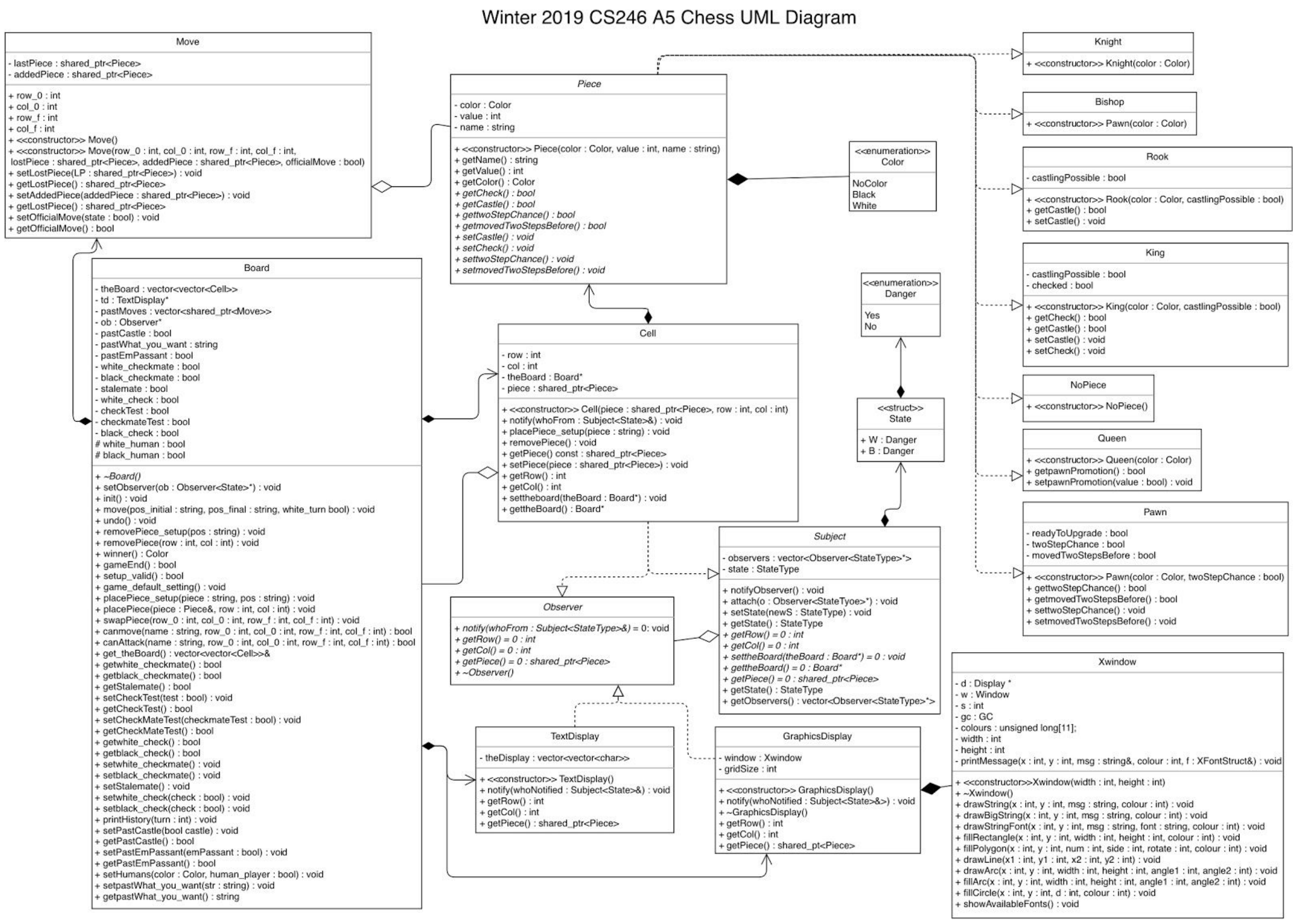
Piece	King	Queen	Rook	Knight	Bishop	Pawn	NoPiece
getName() : string	"king"	"queen"	"rook"	"knight"	"bishop"	"pawn"	"nopiece"
getValue() : int	10*	9	5	3	3	1	0

* King's value is not 10 in an actual game, King is valued 10 so that first priority can be given.

Furthermore, King has an exclusive field: Check, which also makes King possess its corresponding getter and setter: getCheck(), setCheck(). The field is used to determine whether the King Piece is currently under check. In addition to that, the functions related to castling are used by both King and Rook. Lastly, Pawn has member variables and functions to consider the initial two step moves and its potential Pawn promotion.

Finally, we have the **TextDisplay** and **GraphicsDisplay** to draw the board and the dynamic pieces on its static cells. Both of those displays are mostly dependent on the notify() that lets them know which cell has changed and allows to display accordingly.

Updated UML


Winter 2019 CS246 A5 Chess UML Diagram
Move
- lastPiece : shared_ptr<Piece>
- addedPiece : shared_ptr<Piece>
+ row_0 : int
+ col_0 : int
+ row_f : int
+ col_f : int
+ <<constructor>> Move()
+ <<constructor>> Move(row_0 : int, col_0 : int, row_f : int, col_f : int, lostPiece : shared_ptr<Piece>, addedPiece : shared_ptr<Piece>, officialMove : bool)
+ setLostPiece(LP : shared_ptr<Piece>) : void
+ getLostPiece() : shared_ptr<Piece>
+ setAddedPiece(addedPiece : shared_ptr<Piece>) : void
+ getLostPiece() : shared_ptr<Piece>
+ setOfficialMove(state : bool) : void
+ getOfficialMove() : bool
Board
- theBoard : vector<vector<Cell>>
- td : TextDisplay*
- pastMoves : vector<shared_ptr<Move>>
- ob : Observer*
- pastCastle : bool
- pastWhat_you_want : string
- pastEmPassant : bool
- white_checkmate : bool
- black_checkmate : bool
- stalemate : bool
- white_check : bool
- checkTest : bool
- checkmateTest : bool
- black_check : bool
# white_human : bool
# black_human : bool
+ ~Board()
+ setObserver(ob : Observer<State>*) : void
+ init() : void
+ move(pos_initial : string, pos_final : string, white_turn bool) : void
+ undo() : void
+ removePiece_setup(pos : string) : void
+ removePiece(row : int, col : int) : void
+ winner() : Color
+ gameEnd() : bool
+ setup_valid() : bool
+ game_default_setting() : void
+ placePiece_setup(piece : string, pos : string) : void
+ placePiece(piece : Piece&, row : int, col : int) : void
+ swapPiece(row_0 : int, col_0 : int, row_f : int, col_f : int) : void
+ canmove(name : string, row_0 : int, col_0 : int, row_f : int, col_f : int) : bool
+ canAttack(name : string, row_0 : int, col_0 : int, row_f : int, col_f : int) : bool
+ get_theBoard() : vector<vector<Cell>>&
+ getwhite_checkmate() : bool
+ getblack_checkmate() : bool
+ getStalemate() : bool
+ setCheckTest(test : bool) : void
+ getCheckTest() : bool
+ setCheckMateTest(checkmateTest : bool) : void
+ getCheckMateTest() : bool
+ getwhite_check() : bool
+ getblack_check() : bool
+ setwhite_checkmate() : void
+ setblack_checkmate() : void
+ setStalemate() : void
+ setwhite_check(check : bool) : void
+ setblack_check(check : bool) : void
+ printHistory(turn : int) : void
+ setPastCastle(bool castle) : void
+ getPastCastle() : bool
+ setPastEmPassant(emPassant : bool) : void
+ getPastEmPassant() : bool
+ setHumans(color : Color, human_player : bool) : void
+ setpastWhat_you_want(str : string) : void
+ getpastWhat_you_want() : string
Piece
- color : Color
- value : int
- name : string
+ <<constructor>> Piece(color : Color, value : int, name : string)
+ getName() : string
+ getValue() : int
+ getColor() : Color
+ getCheck() : bool
+ getCastle() : bool
+ gettwoStepChance() : bool
+ getmovedTwoStepsBefore() : bool
+ setCastle() : void
+ setCheck() : void
+ settwoStepChance() : void
+ setmovedTwoStepsBefore() : void
<<enumeration>> Color
NoColor
Black
White
<<enumeration>> Danger
Yes
No
<<struct>> State
+ W : Danger
+ B : Danger
Cell
- row : int
- col : int
- theBoard : Board*
- piece : shared_ptr<Piece>
+ <<constructor>> Cell(piece : shared_ptr<Piece>, row : int, col : int)
+ notify(whoFrom : Subject<State>&) : void
+ placePiece_setup(piece : string) : void
+ removePiece() : void
+ getPiece() const : shared_ptr<Piece>
+ setPiece(piece : shared_ptr<Piece>) : void
+ getRow() : int
+ getCol() : int
+ settheboard(theBoard : Board*) : void
+ gettheBoard() : Board*
Subject
- observers : vector<Observer<StateType>*>
- state : StateType
+ notifyObserver() : void
+ attach(o : Observer<StateTyoe>*) : void
+ setState(newS : StateType) : void
+ getState() : StateType
+ getRow() = 0 : int
+ getCol() = 0 : int
+ settheBoard(theBoard : Board*) = 0 : void
+ gettheBoard() = 0 : Board*
+ getPiece() = 0 : shared_ptr<Piece>
+ getState() : StateType
+ getObservers() : vector<Observer<StateType>*>
Observer
+ notify(whoFrom : Subject<StateType>&) = 0: void
+ getRow() = 0 : int
+ getCol() = 0 : int
+ getPiece() = 0 : shared_ptr<Piece>
+ ~Observer()
TextDisplay
- theDisplay : vector<vector<char>>
+ <<constructor>> TextDisplay()
+ notify(whoNotified : Subject<State>&) : void
+ getRow() : int
+ getCol() : int
+ getPiece() : shared_ptr<Piece>
GraphicsDisplay
- window : Xwindow
- gridSize : int
+ <<constructor>> GraphicsDisplay()
+ notify(whoNotified : Subject<State>&>) : void
+ ~GraphicsDisplay()
+ getRow() : int
+ getCol() : int
+ getPiece() : shared_pt<Piece>
Knight
+ <<constructor>> Knight(color : Color)
Bishop
+ <<constructor>> Pawn(color : Color)
Rook
- castlingPossible : bool
+ <<constructor>> Rook(color : Color, castlingPossible : bool)
+ getCastle() : bool
+ setCastle() : void
King
- castlingPossible : bool
- checked : bool
+ <<constructor>> King(color : Color, castlingPossible : bool)
+ getCheck() : bool
+ getCastle() : bool
+ setCastle() : void
+ setCheck() : void
NoPiece
+ <<constructor>> NoPiece()
Queen
+ <<constructor>> Queen(color : Color)
+ getpawnPromotion() : bool
+ setpawnPromotion(value : bool) : void
Pawn
- readyToUpgrade : bool
- twoStepChance : bool
- movedTwoStepsBefore : bool
+ <<constructor>> Pawn(color : Color, twoStepChance : bool)
+ gettwoStepChance() : bool
+ getmovedTwoStepsBefore() : bool
+ settwoStepChance() : void
+ setmovedTwoStepsBefore() : void
Xwindow
- d : Display *
- w : Window
- s : int
- gc : GC
- colours : unsigned long[11];
- width : int
- height : int
- printMessage(x : int, y : int, msg : string&, colour : int, f : XFontStruct&) : void
+ <<constructor>>Xwindow(width : int, height : int)
+ ~Xwindow()
+ drawString(x : int, y : int, msg : string, colour : int) : void
+ drawBigString(x : int, y : int, msg : string, colour : int) : void
+ drawStringFont(x : int, y : int, msg : string, font : string, colour : int) : void
+ fillRectangle(x : int, y : int, width : int, height : int, colour : int) : void
+ fillPolygon(x : int, y : int, num : int, side : int, rotate : int, colour : int) : void
+ drawLine(x1 : int, y1 : int, x2 : int, y2 : int) : void
+ drawArc(x : int, y : int, width : int, height : int, angle1 : int, angle2 : int) : void
+ fillArc(x : int, y : int, width : int, height : int, angle1 : int, angle2 : int) : void
+ fillCircle(x : int, y : int, d : int, colour : int) : void
+ showAvailableFonts() : void

Design

Our final design differs from the old UML (Due Date 1) in various ways even though the basic concept still remains equivalent. Like previously mentioned, we used the Observer pattern to take care of the interaction between cells and the movements of pieces that take place.

However, there are many more member variables and methods compared to the old program design. There are various booleans and functions added to take care of the specifics such as castling, en-passant, or to determine if the King is under check or checkmate. Our team had to add a new class, Move, to get the last official move so that the program can correctly take care of check positions, using undo() from the Board class. The function along with the Move class also allowed us to implement one of the extra features that helps us undo multiple moves made by the players.

In terms of using observer pattern, it was used to implement the interactions between classes because there are many one-to-many relationships in the program Chess. In the game of Chess, the player moves a piece from a certain cell to another. In the process, the cells' locations are static, they are always on their original assigned coordinates on the board. In the meantime, the pieces are dynamic as they move around the board and place themselves from cell to cell.

We assigned the text and graphics display classes as observers, and the cells are both observers and subjects to each others. Specifically, the text display and graphic display classes observe all 64 cells on the board; therefore, when the cell changes its state, the display classes update their data based on the cell's notification. In terms of lower classes, each cell observes all other 63 cells, excluding itself. As a result of each player's movement, the state of the cell at which its piece has changed changes. The changed state of the cell notifies its observers, the other 63 cells, and hence the notification on each cell changes the state of the board. In addition, each cell also notifies the text and graphic displays so that there occurs a change on the user interface.

Observer pattern is quite useful in the game of Chess due to mainly two reasons:

1. Less coupling between objects that interact with each other.
   With the implementation of notifying the observer function, and the interface of receiving notification from the subject using template, each cells, text display, and graphics display possess their own method of receiving and updating their states. With unified way of notifying the observers for the subjects, there is not much coupling between objects.
2. Simple object attach / remove process
   The association between observer and subject can be easily added or removed using unified attach function for the subject. Via unification method of attaching observers to each subject, the relationship between objects does not have to individually handled in the game flow implementation.

The Computer Players

Our team deployed 4 levels of AI players. These players are not perfect like the online chess AI but they implement the required strategy of moves in accordance with the assignment guidelines. The four levels are carried out as follows:

Level 1:

This level aims to make random but valid moves for the pieces. Our team has first defined a vector to save pointers to the different Pieces on the board. Then a piece is randomly chosen from the vector by generating a random number using rand() from standard C++ <cstdlib> library; specifically, the random number does not exceed the number of total pieces present in the vector. The program once again tries to obtain random coordinates for the chosen piece to move to. If the specific piece has a valid move to the chosen position, then the move is carried out by calling move() in the Board class. If none of the random move works then the program tries to force a valid move. The program checks the validity of the move using canmove() in both cases.

Level 2:

At level 2, the program prioritizes capturing the opponent's pieces over randomness, followed by attempting to check the opponent's King before resorting to the functionality of level 1. Initially, the program uses two vectors: MyPieces vector for storing pointers to all the white pieces if it's white's turn and OpPieces vector for the black ones and vica versa. Then to make a capture move, the program first tries to locate the opponent's piece with the highest value. The first valid move to those coordinates using any of the pieces in MyPieces is preferred. If impossible, then the program tries to capture the one with the next highest value. Once all capturing attempts have failed, each piece based on its type tries to make a valid move such that it puts the opponent's king under check. If all the pieces fail to do so, then level 2 AI player calls level 1 and tries to make a random valid move.

Level 3:

Level 3 follows instructions such that avoiding capture is the first preference. If the state of all the cells on which one's pieces are on are not in danger, then the program follows the preferences in level 2. To avoid capture, all pointers to the pieces that are in danger are stored in a vector. Afterwards, the program tries to save the piece starting from the highest value. If it is impossible to save that particular piece then we move on to the piece with the next highest value, and so on. If a king is in danger, the program tries to make any possible move that brings the king out of check.

Level 4:
The last level's behaviour is similar to that of level 2. However, the key difference is that it only tries to capture the opponent's piece if that capture move does not put one's piece in danger after it's carried out. As a result, this level tries to carry out a more cautious approach. Hence, the next preference is to avoid getting captured if it is possible. Thus if a piece cannot capture while ensuring it's safety, then level 3 AI player is called.

After multiple simulations of computer versus computer games, our team realized that the levels with a more offensive approach have a greater tendency of winning. While level 4 is the most powerful AI player among all, level 3 is surprisingly not guaranteed to win against level 2.

Resilience to Change

Our team tried to make the modules and functions in such a way that we would not require a complete overhaul of our program to accommodate change in features, design or rules.

For example, if we have to change the input format then all we have to do is change the implementation of some helper functions: valid_pos(), row_return(), and col_return() in board.cc. As the program embraces our team's own interpretation of row and column coordinates, all we need to do is to implement a conversion from new input format to our way of representing the row and columns.

If we have to replace the type of a piece or add a different set of rules for a certain piece, then instead of changing everything, we only need to change the member variables and functions for that specific piece, and change the implementation rules in functions such as canmove() in board.cc. This will also not require a lot of dedication because each piece has been taken care in different blocks of the function code.

Our program can also easily add new features other than castling or en passant. All we need to is to add or to remove the member variables and methods for the pieces that are involved and write an additional condition in move() and canmove() for that feature to be added in our implementation.

Additionally, adding more computer levels is simple for our program. As our program possesses separate computer.h/.cc files, we will need to define a new level for the AI and add an additional command interpreter in our main.cc file. In fact, the earlier levels of AI players can be applied to the new level AI algorithm since each AI is deployed in the program as a function; specifically, each offensive, defensive, and other strategic moves can be drawn from the earlier levels since they possess their own traits of moving pieces.

Finally, we can also make different variants of chess using our program. The answer to question 3 in the next section lists the functions that we will have to make changes to make a four-handed chess game. Without much disruption to the program, it is fairly straightforward to make a completely new game by slightly changing the code that we have already written.

Now in terms of cohesion, our program is composed of high cohesiveness as it possesses every individual module for each class, with each class serving its own unique purpose. We tried to achieve functional cohesion in our program by grouping together modules according to the specific tasks they are responsible for. Each of our classes work towards their own unique goal such as the **Piece** class is responsible for storing the information about chess pieces. Private fields such as Name, Color and Value pertain to a piece's information and both getter and setter have been implemented for each field in piece class. **Cell** class is responsible for storing the information of each cell on the chess board. Hence fields such as row, col and piece provide the program with the necessary information about cell and therefore, belong solely in the cell class. Accordingly, getter and setter methods for each field have been implemented in the cell class. **Board** class stores the information pertaining to the state of the game such as if any of the kings have been put in check, checkmate or if the game has ended with a stalemate. All this information is stored in various field such as white_checkmate, black_checkamte, white_checkmate, black_checkmate and stalemate. All these fields are private because the state of the game should be changed in board object only as all the other classes such as **Piece** and **Cell** work towards their own goal. Hence, by segregating the classes according to their workflow, we were able to achieve high cohesion.

Our program follows in line with low coupling as the modules only communicate with each other using function calls that return simple results. For example board.cc uses getColor() and getName() functions from piece.h that return simple string or Color values. Changes in one module don't affect the other modules. For example, if we have to change piece and maybe add another type to it then we don't have to make changes to the other .h files.

As our program Chess involves a lot of information, encapsulation of data was also crucial. The program possesses every individual module for each class, with each class serving its own unique purpose with pointers to its associated classes. For instance, the highest module **Board** class possesses a vector of vector of class **Cell**. Correspondingly, each **Cell** has a pointer to an instance of class **Piece** that is currently located on its coordinates. In addition, with **TextDisplay** and **GraphicsDisplay** being part of the Board composition as Observers, they interact with the Cell updates embracing it as a Subject with the list of its own Observers. With such robust encapsulation of variables using appropriate classes, the data can be easily handled in a secured way.

Answers to Questions

Answer to question 1 essentially remains the same, but there are some tweaks made so as to adjust it to the current functionality of our program. Question 2 is answered differently from DD1 as our team was able to implement that specification but utilized a different method to carry it out. The answer to the last question remains unchanged.

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.

To tackle this problem, we'll use a map structure, **OpeningMoves** in our class Board. The key part of the structure will store the name of the opening move; for example, "Two Knights Defense" and then the program has another Map structure for the value part of it. The "value" map structure will contain a specific Text_Display of the board in the key part and respective next move based upon the composition of the board. We can store all our opening moves this way and these will be immutable and be fixed structures for the moves. To implement the correct counter-move in the opening sequence, we'll create a Cell_compare function in our class board so that we can compare two board representations using it. After every command move from the other player, we'll iterate over the given opening sequences' Map structure and compare it to the current state of our board and use the value part to get the coordinates for our next move.

Question 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?

We created a vector of **Move** class, pastMoves in the board class. The move class has fields such as coordinates of the initial move, final move, addedPiece, lostPiece and officialMove. In chess, there are multiple types of move which can take place such as capturing a piece, pawn promotion, En-Passant and castling. In order to implement the undo function for all kinds of moves, we used addedPiece field to store the promoted piece when the pawn would be in the state to get promoted, losePiece field was used to store the piece which was lost in attack such that whenever the player calls undo, the lostPiece can be restored back to its old position and officialMove field was used to differentiate between the official move taken by the player and the move which would take place in correspondence to the official move. For example, when a player makes a castling move, movement of the king would be an an official move whereas movement of rook would be an unofficial move. By accounting for all these moves, we were able to successfully store the required information about every move. Therefore, when a player inputs "undo" in the command line, the last move element is popped from pastMoves using pastMoves.pop_back() and the popped move object is used to change the state of the board accordingly. Undo command can be called multiple times by iterating through the pastMoves vector until pastMoves.empty() returns true.

This answer differs from the answer submitted for DD1 in the following ways. DD1's answer mentioned that the vector would be made up of another vector which would store the coordinates of the move, but in our program a vector of Move class was used. DD1's answer did not mention how different kinds of moves would be accommodated and where the lost and new pieces would be stored.

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.

Beginning from our class Board, we'll first need to add an option to make a team so that if two players want then they can plan together. Secondly, we'll need to modify our functions such that the game represents a four-player chess board and moves instead of the original one.
For instance, our function that creates the board init() will now have 12 instead of 8 associated vectors and the middle 8 of them will have 12 instead of 8 cells. Next, we will change our game_default_setting() method and add another two set of pieces on the specified positions for the other two players. After that, setOberver() will be changed accordingly to accommodate the additional cells and pieces; in addition, winner() and gameEnd() methods will change correspondingly as well. Finally, gameEnd() will be changed such that three Kings need to be checkmated before the fourth can be declared as the winner, and two Kings of the same team for the other team to win in order for the game to end. The specifications of the **TextDisplay** and **GraphicsDisplay** will be changed to represent the new board.

Extra Credit Features

- Undo: We implemented the history feature by adding a command interpreter in the main function which reads "undo" from the input stream. Implementing the undo function was quite challenging as we had to account for different types of moves possible in a game of chess. Capture moves would remove a piece from the board and in order to undo the capture move, the removed would have to be stored somewhere and hence we decided to create a Piece pointer field that will store the lost piece. Additionally, castling comprises two moves, hence the initial and final position of rook had to be stored as well in order to undo a castling move. Therefore, we decided to create a boolean field called OfficialMove which would differentiate between a player's official move and the corresponding move related to it.

- History: We implemented the history feature by adding a command interpreter in the main function which reads "history" from the input stream. Implementing the undo functionality first made it easy for us to print out all the moves taken by the players in a chess game by iterating over the pastMoves vector until pastMoves.empty() returned true. OfficialMove field allowed us to differentiate between the official move of the player and corresponding move to it.
- Graphics: With board and piece designs designed by the team using X11 commands, the graphics display represents not only the actual movements made by the user or AI player, but also displays the recommend move, and even the flow of the AI player's thought process.

Final Questions

Question 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?

Most importantly, we learnt that developing software in teams requires cooperation, communication, and effective collaboration. It was not always easy to follow the train of thought of the other team members, but once we got used to it and spent time on understanding what the other person wants to do, we found that the implementation became way easier. Furthermore, this process helped us in terms of eradicating errors or missed cases from earlier times. As everyone has different schedules, we had to make sure we were communicating regularly and letting each other know about the progress. To prevent any contradictory code, we tried to make sure to work at different files. However, it was difficult to maintain that sort of separation all the time. Ultimately, git allowed us to share our changes once everyone was done with their work and in turn take care of merge errors if there were any.

Question 2:

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

The only thing we would have done differently is change the order in which we implemented our functions. We would implement the notify function in our Observers before anything else as most of our errors later were dependent on or related to that. Additionally, we wrote check and checkmate after we were done with most of our AI levels. Maybe, it would have been better if we had reversed the order for that.