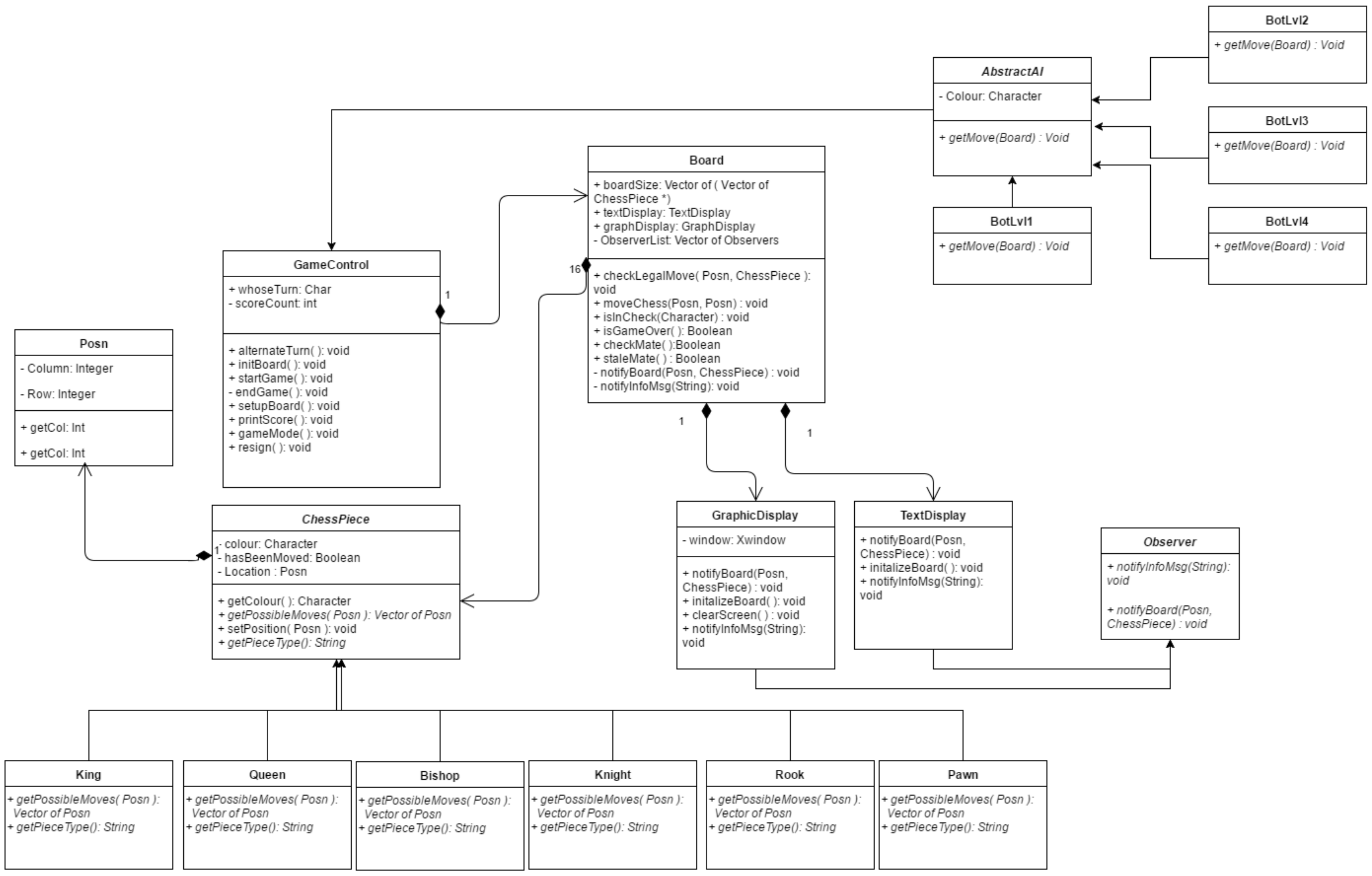
UML


BotLvl2
+ getMove(Board) : Void
AbstractAI
- Colour: Character
+ getMove(Board) : Void
BotLvl3
+ getMove(Board) : Void
BotLvl1
+ getMove(Board) : Void
BotLvl4
+ getMove(Board) : Void
Board
+ boardSize: Vector of (Vector of ChessPiece *)
+ textDisplay: TextDisplay
+ graphDisplay: GraphDisplay
- ObserverList: Vector of Observers
+ checkLegalMove(Posn, ChessPiece): void
+ moveChess(Posn, Posn) : void
+ isInCheck(Character) : void
+ isGameOver( ): Boolean
+ checkMate( ):Boolean
+ staleMate( ) : Boolean
- notifyBoard(Posn, ChessPiece) : void
- notifyInfoMsg(String): void
16
1
GameControl
+ whoseTurn: Char
- scoreCount: int
+ alternateTurn( ): void
+ initBoard( ): void
+ startGame( ): void
- endGame( ): void
+ setupBoard( ): void
+ printScore( ): void
+ gameMode( ): void
+ resign( ): void
Posn
- Column: Integer
- Row: Integer
+ getCol: Int
+ getCol: Int
1
1
1
ChessPiece
- colour: Character
- hasBeenMoved: Boolean
- Location : Posn
+ getColour( ): Character
+ getPossibleMoves(Posn): Vector of Posn
+ setPosition(Posn): void
+ getPieceType(): String
GraphicDisplay
- window: Xwindow
+ notifyBoard(Posn, ChessPiece) : void
+ initalizeBoard( ): void
+ clearScreen( ) : void
+ notifyInfoMsg(String): void
TextDisplay
+ notifyBoard(Posn, ChessPiece) : void
+ initalizeBoard( ): void
+ notifyInfoMsg(String): void
Observer
+ notifyInfoMsg(String): void
+ notifyBoard(Posn, ChessPiece) : void
King
+ getPossibleMoves(Posn): Vector of Posn
+ getPieceType(): String
Queen
+ getPossibleMoves(Posn): Vector of Posn
+ getPieceType(): String
Bishop
+ getPossibleMoves(Posn): Vector of Posn
+ getPieceType(): String
Knight
+ getPossibleMoves(Posn): Vector of Posn
+ getPieceType(): String
Rook
+ getPossibleMoves(Posn): Vector of Posn
+ getPieceType(): String
Pawn
+ getPossibleMoves(Posn): Vector of Posn
+ getPieceType(): String

Plan of Attack

Project Timeline

- July 13 – Planning stage
    - Pattern Design, UML
    - Classes_functions.xls (a document describes what the functions are expected to behave by plain words)
- July 15 – Setup stage
    - Everyone will initiate the .cc and .h files for each class. This creates basic setup of the program and speeds up the development process.
- July 20 – Implement stage
    - Bo will implement ChessPiece (and all inherited classes), Posn, GameControl, and game history.
    - Chris will implement TextDisplay then the AI.
    - Chuck will implement Board, GraphicDisplay, and Observer.
- July 21 - 23 – Test stage
- July 23 – Advancements
    - Implementation of undoing any chess move by Bo.
    - Main Function by Chris/Chuck.
- July 25 - Final Design Brief to be completed by all group members.

Implementation of 4 player chess option if time allows. Each group member is responsible for adding necessary modifications to their classes they created originally. Group members will be required to assist each other should any finish early.

Implement Stage

Our code needs to be high cohesion and low coupling. We decided to implement with Model-View-Controller and Observer pattern in our chess game. If we do so, there exists a convenient way to make changes to our program and implement additional features into our chess game without rewriting large amount of code

GameControl is the controller; Board & ChessPiece are the Model; View is the Display parts. The GameControl manipulates the Board. The Board and ChessPiece directly manage the data logic and rules of application. Display and Observer is the representation of information. GameControl accepts user input and feeds it to the model or view as appropriate.

The Observer classes in our program are notified when there are changes to the board or to the messages that should be shown to the players.

Hope for the best, prepare for the worst:

Just in case, we prioritized some segments (from most important to least important):

- The Board and Chess Piece: functionality & text display
- The logic of chess piece movement: allow pieces to move correctly in the board
- The flow of the game: distinguishing players, controlling turns, start/end game, score
- Advanced rules: en passant, castling, etc.
- AI: level one to three
- The rest of the classes and functions in UML & requirements specified in guidelines
- Graphic display and level four AI
- Bonus: additional features for the game

This will gives us a metric on how to sequence our implementations for the actual program.

<u>Test stage</u>

After we have the board and chess pieces being represented in the text display setup, we can start to test the code.

- Test all chess pieces: make sure it follows the game logic.
- Integration test, implement the flow of the game with the chess and board to make it a proper game.
- Test for advanced chess rules.
- Test AI
- Integration test, test the entire game runs, throw no exceptions.
- Invite chess players to test code by playing it.
- After setup of the graph Display, test it.

We will do black box and white box testing. It is beneficial to test each other's code because different people have different point of views and ideas. We decided to test the code together in meetings.

<u>Chess Questions</u>

Question:

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.

Answer:

Create a StandardOpening class with the following fields:

- Stack of moves in sequential order (which will represent a standard opening if played)

Create a StandardOpeningBook with:

- Vector containing all desired StandardOpenings
- A method to check if the last move made corresponds to any StandardOpening

After every move, a check will be run via the Book's (StandardOpeningBook's) method.

This check will pop top move off of each StandardOpening's stack and compare it to the actual move in game. If the move is the same, that StandardOpening is preserved and the next move in the stack could be applied by an AI (depending on AI). There is a possibly that multiple different StandardOpenings would apply and the AI would then need to choose between possible moves. If the move is not the same this openings is removed from StandardOpeningBooks vector of StandardOpenings.

Question:

How would you implement a feature that would allow a player to undo his/her last

move? What about an unlimited number of undos?

Answer:

Create a CMove class with the following fields:

- Original coordinates of chess piece that was moved
- New coordinate of chess piece that was moved
- Colour of chess piece that was moved
- Chess piece that was captured (if any)
- Castling flag
- Capture en passant flag

Create a History class with:

- A stack of CMove objects as a private field within
- An undo Method that pops a move off the stack and reverses the last move on the board

-After every move, a corresponding CMove object will be pushed onto the History object's stack

-If the user indicates they want an undo (presumably through typing a command "undo") the History object's undo method will respond appropriately.

-If no moves have been made, undo does nothing

-Last moved piece will return to its prior position

-If the last move was a castling move, logic for moving the king and rook back to their appropriate positions is run

-If the last move was an en passant capture, special logic for replacing the captured pawn is run

-If the last move was a normal capturing move, the piece that was removed is replaced at the spot the moved piece is returning from

-If a book of openings is being used by AI, the AI would shift one move back.

In the case that we are not allowing unlimited undoes (only last move can be undone), there would be no need for a stack, the history class would store only one move as a field. All the other logic would remain similar.

Question:

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.

Answer:

The current board implementation is a vector (theBoard) that contains 8 vectors (theRows) which each contain 8 piece pointers (theColumns).Additional rows would simply be created by pushing additional row vectors to the board. Rows are customizable and need not all contain the same number of columns.

In the context of four handed chess:

- theBoard would contain 3 rows of 8 columns followed by
    - 8 rows of 14 columns, followed by 3 rows of 8 columns.
- Pieces would be created and pointed to as appropriate on this custom board
- Pieces would be able to be assigned to any of four different colour allegiances
- Turn order would be passed in the appropriate order.
- Win condition would have to be changed appropriately
- Assuming allegiances exist between players could exist, conditions for check
    - would have to change from any piece of other color being able to take attack ones king, to any piece from opposing team being able to attack ones king.
    - Capture logic would also need to be altered similarly.
- Pawn promotion logic would need to be altered
- AI would need to be overhauled